R for Non-Programmers: A Guide for Social Scientists

Daniel Dauber

2021-08-05

Contents

Welcome

Welcome to *R for Non-Programmers: A guide for Social Scientists*. This book is intended to be of help to everyone who wishes to enter the world of R Programming, but not necessarily for the purpose to become a programmer. Instead, this book intends to convey key concepts in data analysis, especially quantitative research.

[Add more]

Acknowledgments

Special thanks are due to my wife, who supported me in so many ways to get this book completed. Also, I would like to thank my son, who patiently watched me sitting at the computer type this book up.

Chapter 1

`Readme.` Before you get started

1.1 A starting point and reference book

`to be added`

1.2 Download the companion R package

`to be added`

- Explain where to download
- Explain how examples are shown in this book (code in the first grey box, console results in the second grey box

1.3 A 'tidyverse' approach with some basic R

`to be added`

Chapter 2

Why learn a programming language as a non-programmer?

'R', it is not just a letter you learn in primary school, but a powerful programming language. While it is used for a lot of quantitative data analysis, it has grown over the years to become a powerful tool that excels (#no-pun-intended) in handling data and performing customised computations with quantitative and qualitative data.

R is now one of my core tools to perform various types of analysis because I can use it in many different ways, for example,

- statistical analysis,
- corpus analysis,
- development of online dashboards to dynamically generate interactive data visualisations,
- connection to social media APIs for data collection,
- Creation of reporting systems to provide individualised feedback to research participants,
- Drafting and writing research articles, etc.

> *Learning R is like learning a foreign language. If you like learning languages, then 'R' is just another one.*

While *R* has become a comprehensive tool for data scientists, it has yet to find its way into the mainstream field of Social Sciences. Why? Well, learning programming languages is not necessarily something that feels comfortable to

everyone. It is not like Microsoft Word, where you can open the software and explore it through trial and error. Learning a programming language is like learning a foreign language: You have to learn vocabulary, grammar and syntax. Similar to learning a new language, programming languages also have steep learning curves and require quite some commitment.

For this reason, most people do not even dare to learn it because it is time-consuming and often not considered a *'core method'* in Social Sciences disciplines. Apart from that, tools like SPSS have very intuitive interfaces, which seem much easier to use (or not?). However, the feeling of having *'mastered' R* (although one might never be able to claim this) can be extremely rewarding.

I guess this introduction was not necessarily helpful in convincing you to learn any programming language. However, despite those initial hurdles, there are a series of advantages to consider. Below I list some good reasons to learn a programming language as they pertain to my own experiences.

2.1 Learning new tools to analyse your data is always essential

Theories change over time, and new insights into certain social phenomena are published every day. Thus, your knowledge might get outdated quite quickly. This is not so much the case for research methods knowledge. Typically, analytical techniques remain over many years. We still use the mean, mode, quartiles, standard deviation, etc., to describe our quantitative data. Still, there are always new computational methods that help us to crunch the numbers even more. *R* is a tool that allows you to venture into new analytical territory because it is open source. Thousands of developers provide cutting-edge research methods free of charge for you to try with your data. You can find them on platforms like GitHub. *R* is like a giant supermarket, where all products are available for free. However, to read the labels on the product packaging and understand what they are, you have to learn the language used in this supermarket.

2.2 Programming languages enhance your conceptual thinking

While I have no empirical evidence for this, I am very certain it is true. While I would argue that my conceptual thinking is quite good, I would not necessarily say that I was born with it. Programming languages are very logical. Any error in your code will make you fail to execute it properly. Sometimes you face challenges in creating the correct code to solve a problem. Through creative abstract thinking (I should copyright this term), you start to approach your problems differently, whether it is a coding problem or a problem in any other

context. For example, I know many students enjoy the process of qualitative coding. However, they often struggle to detach their insights from the actual data and synthesise ideas on an abstract and more generic level. Qualitative researchers might refer to this as challenges in '*second-order deconstruction of meaning'*. This process of abstraction is a skill that needs to be honed, nurtured and practised. From my experience, programming languages are one way to achieve this, but they might not be recognised for this just yet.

2.3 Programming languages allow you to look at your data from a different angle

There are certainly commonly known and well-established techniques regarding how you should analyse your data rigorously. However, it can be quite some fun to try techniques outside your disciplines. This does not only apply to programming languages, of course. Sometimes, learning about a new research method enables you to look at your current tools in very different ways too. One of the biggest challenges for any researcher is to reflect on your work. Learning new and maybe even *'strange'* tools can help with this. Admittedly, sometimes you might find out that some new tools are also a dead-end. Still, you might have learned something valuable through the process of engaging with your data differently. So shake off the rust of your analytical routine and blow some fresh air into your research methods.

2.4 Learning any programming language will help you learn other programming languages.

Once you understand the logic of one language, you will find it relatively easy to understand new programming languages. Of course, if you wanted to, you could become the next '*Neo'* (from 'The Matrix') and change the reality of your research forever. On a more serious note, though, if you know any programming language already, learning R will be easier because you have accrued some basic understanding of these particular types of languages.

Having considered everything of the above, do you feel ready for your next foreign language?

Chapter 3

Setting up R and RStudio

Every journey starts with gathering the right equipment. This intellectual journey is not much different. The first step that every 'R novice has to face is to set everything up to get started. There are essentially two strategies:

- Install *R*and RStudio

or

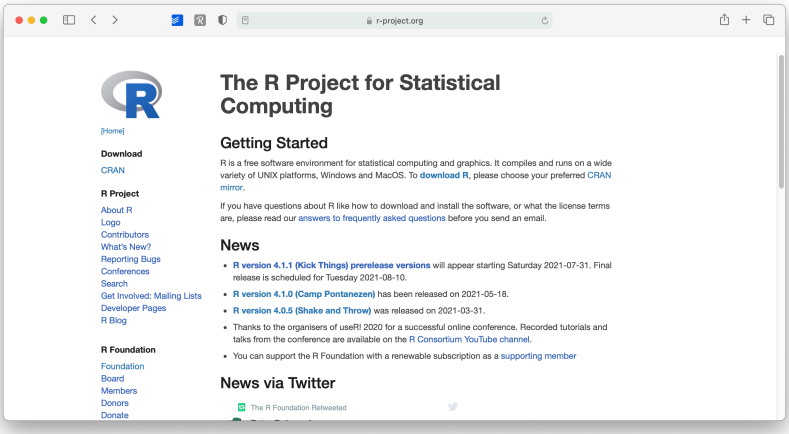
- Run RStudio in a browser via RStudio Cloud

While installing *R* and Studio requires more time and effort, I strongly recommend it, especially if you want to work offline or make good use of your computer's CPU. However, if you are not sure yet whether you enjoy learning *R*, you might wish to look at RStudio Cloud first. Either way, you can follow the examples of this book no matter which choice you make.

3.1 Installing R

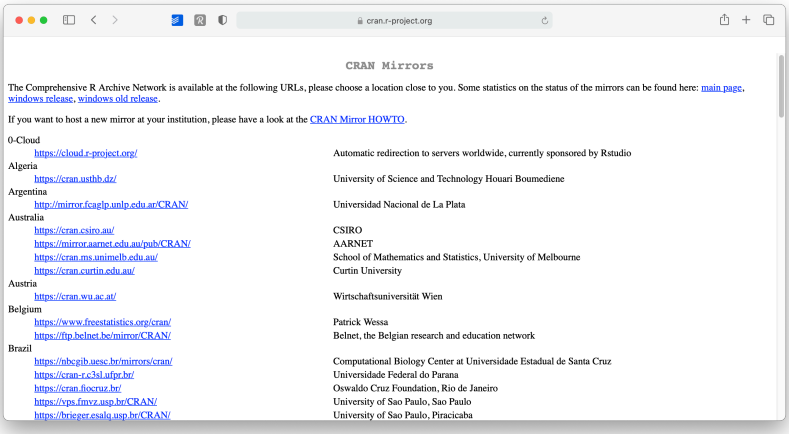
The core module of our programming is R itself, and since it is an open-source project, it is available for free on Windows, Mac and Linux computers. Here is what you need to do to install it properly on your computer of choice:

1. Go to www.r-project.org

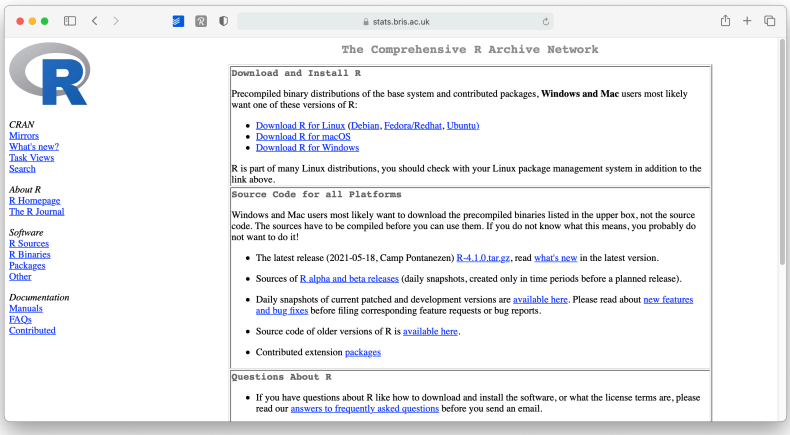


2. Click on `CRAN` where it says download.

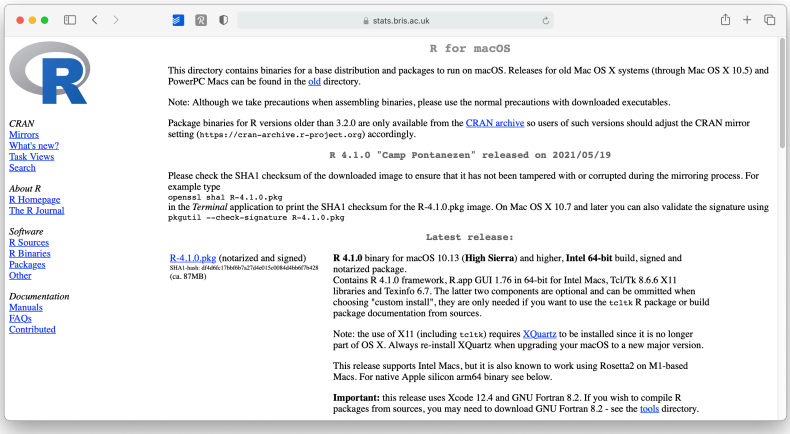
3. Choose a server in your country (all of them work, but downloads will perform quicker).



4. Select the operating system for your computer.



5. Select the version you want to install (I recommend the latest version)



6. Open the downloaded file and follow the installation instructions. (I recommend leaving the suggested settings as they are).

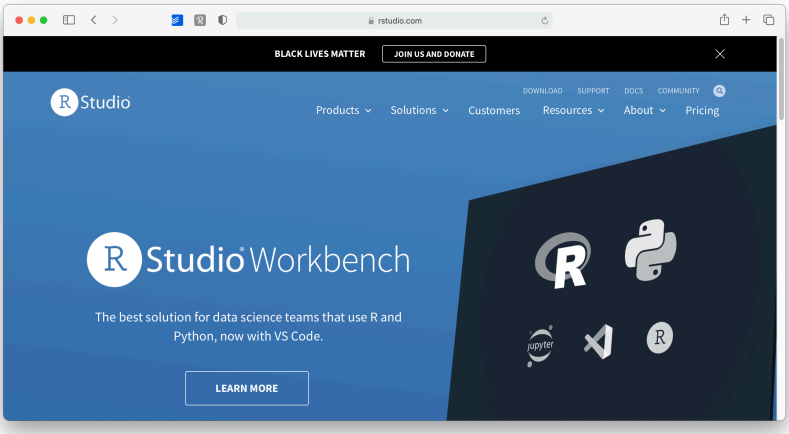
This was relatively easy. You now have *R* installed. Technically you can start using *R* for your research, but there is one more tool I strongly advise installing: RStudio.

3.2 Installing RStudio

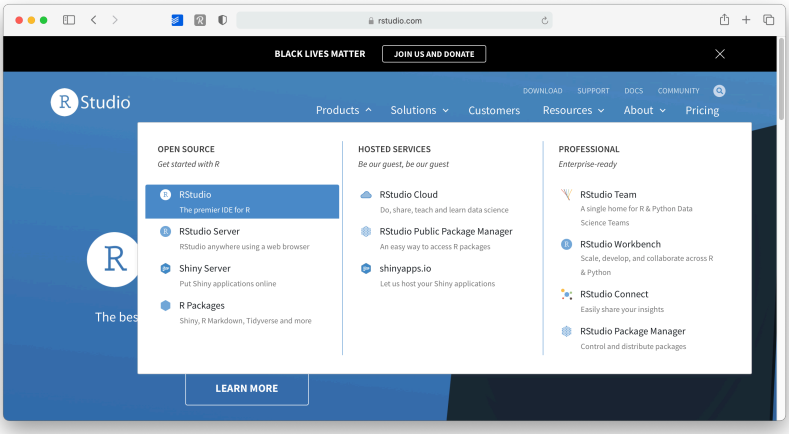
R by itself is just the *'beating heart'* of *R* programming, but it has no particular user interface. If you want buttons to click and actually 'see' what you

are doing, there is no better way than RStudio. RStudio is an *integrated development environment* (IDE) and will be our primary tool to interact with *R*. It is the only software you need to do all the fun parts and, of course, to follow along with the examples of this book. To install RStudio perform the following steps:

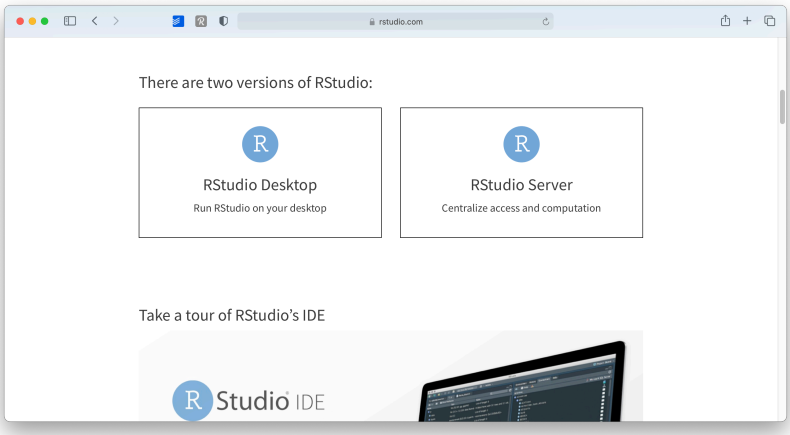
1. Go to www.rstudio.com.



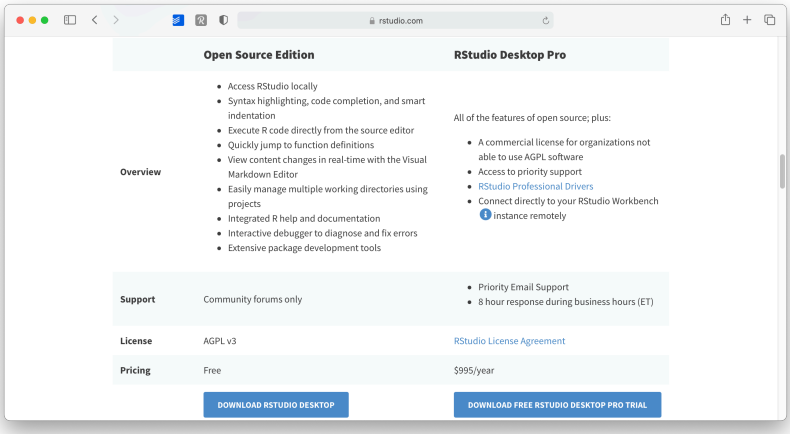
2. Go to `Products > RStudio`



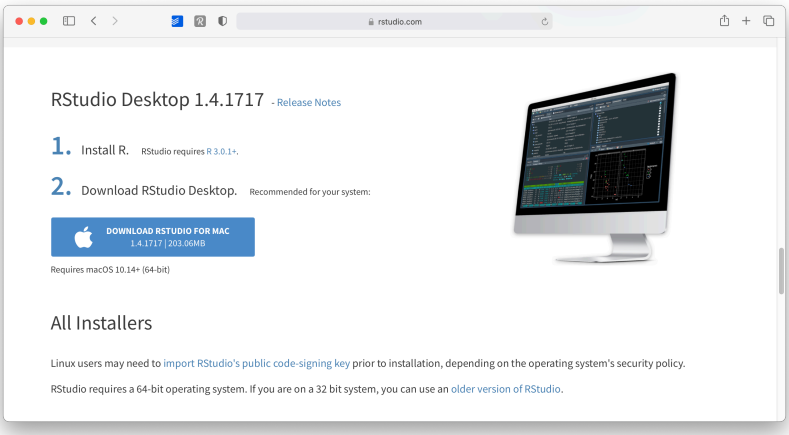
3. On this page, scroll down and select `RStudio Desktop`



4. Select the `'Open Source Edition'` option by clicking on `'Download RStudio Desktop'`



5. As a last step, scroll down where it shows you a download button for your operating system. The website will automatically detect this. You also get a nice reminder to install 'R' first, in case you have not done so yet.



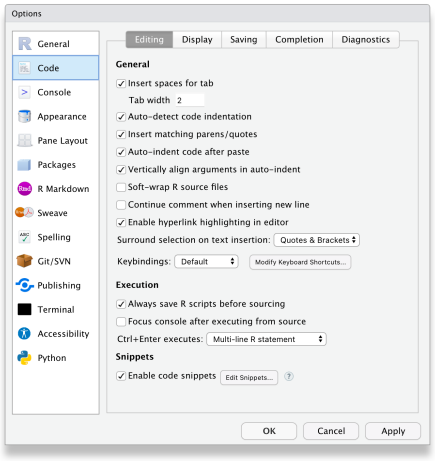
6. Open the downloaded file and follow the installation instructions (again, keep it to the default settings as much as possible)

Congratulations, you are all set up to learn *R*. From now on you only need to start RStudio and not *R*. Of course, if you are the curious, nothing shall stop you to try *R* without RStudio.

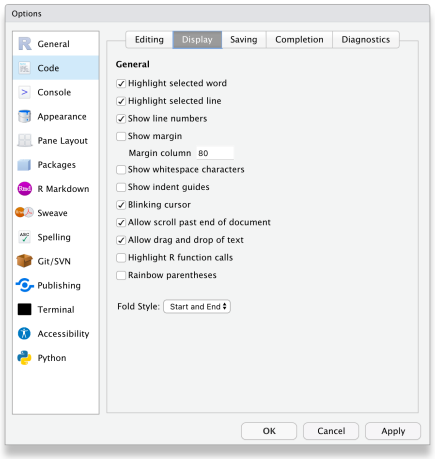
3.3 When you first start RStudio

Before you start programming away, you might want to make some tweaks to your settings right away to have a better experience (in my humble opinion). I recommend at least the following two changes by clicking on `RStudio > Preferences` or press `/Ctrl + ,`.

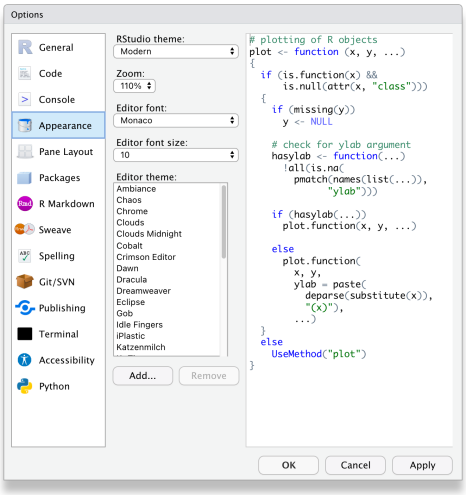
1. In the `Code > Editing` tab, make sure to have at least the first five options ticked, especially the `Auto-indent code after paste`. This setting will save time when trying to format your coding appropriately, making it easier to read. Indentation is the primary way of making your code look more readable and less like a series of characters that appear almost random.



2. In the `Display` tab, you might want to have the first three options selected. In particular, `Highlight selected line` is helpful because, in more complicated code, it is helpful to see where your cursor is.



Of course, if you wish to customise your workspace further, you can do so. The visually most impactful way to alter the default appearance of RStudio is to select Appearance and pick a completely different colour theme. Feel free to browse through various options and see what you prefer. There is no right or wrong here.



3.4 Updating R and RStudio: Living at the pulse of innovation

While not strictly something that helps you become a better programmer, this advice might come in handy to avoid turning into a frustrated programmer. When you update your software, you need to update R and RStudio separately from each other. While both R and RStudio work closely with each other, they still constitute separate pieces of software. Thus, it is essential to keep in mind that updating RStudio will not automatically update R. This can become problematic if specific packages you installed via RStudio (like a fancy learning algorithm) might not be compatible with earlier versions of R. Also, additional R packages developed by other people are separate pieces and are updated too, independently from R and RStudio.

I know what you are thinking: This already sounds complicated and cumbersome. However, rest assured, we take a look at how you can easily update all your packages with RStudio. Thus, all you need to remember is: *R* needs to be updated separately from everything else.

3.5 RStudio Cloud

`to be completed`

Chapter 4

The RStudio Interface

The RStudio interface is composed of quadrants, each of which fulfils a unique purpose:

- The `Console` window,
- The `Source` window,
- The `Environment / History / Connections / Tutorial` window, and
- The `Files / Plots / Packages / Help / Viewer` window

You might only see three windows and wonder where the `Source` window has gone in your version of RStudio. In order to use it you have to either open a file or create a new one. You can create a new file by selecting `File > New File > R Script` in the menu bar, or use the keyboard shortcut `Ctrl+Shift+N` on PC and `Cmd+Shift+N` on Mac.

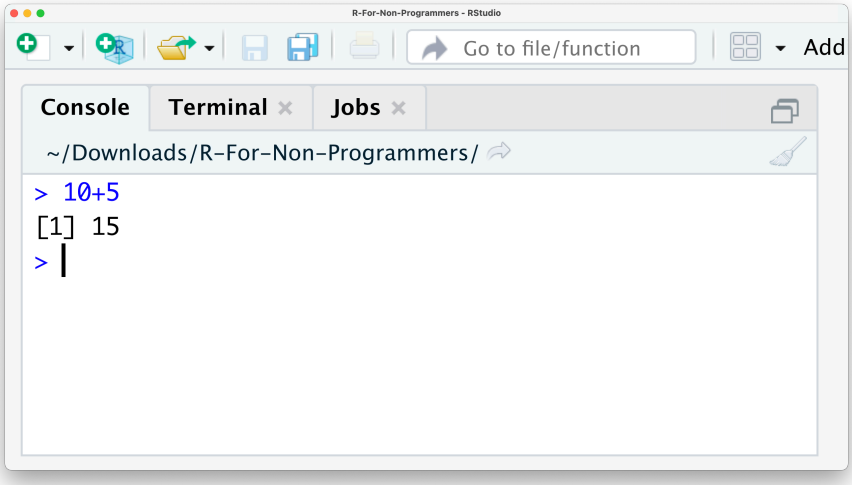
I will briefly explain the purpose of each window/pane and how they are relevant to your work in *R*.

4.1 The Console window

The console is located in the bottom-left, and it is where you often will find the output of your coding and computations. It is also possible to write code directly into the console. Let's try the following example by calculating the sum of `10 + 5`. Click into the console with your mouse, type the calculation into your console and hit `Enter/Return` on your keyboard. The result should be pretty obvious:

```
# We type the below into the console
10+5
## [1] 15
```

Here is a screenshot of how it should look like at your end in RStudio:



You just successfully performed your first successful computation. I know, this is not quite impressive just yet. *R* is undoubtedly more than just a giant calculator.

In the top right of the console, you find a symbol that looks like a broom. This one is quite an important one because it clears your console. Sometimes the console can become very cluttered and difficult to read. If you want to remove whatever you computed, you can click the broom icon and clear the console of all text. I use it so frequently that I strongly recommend learning the keyboard shortcut, which is `Ctrl+L` on PC and Mac.

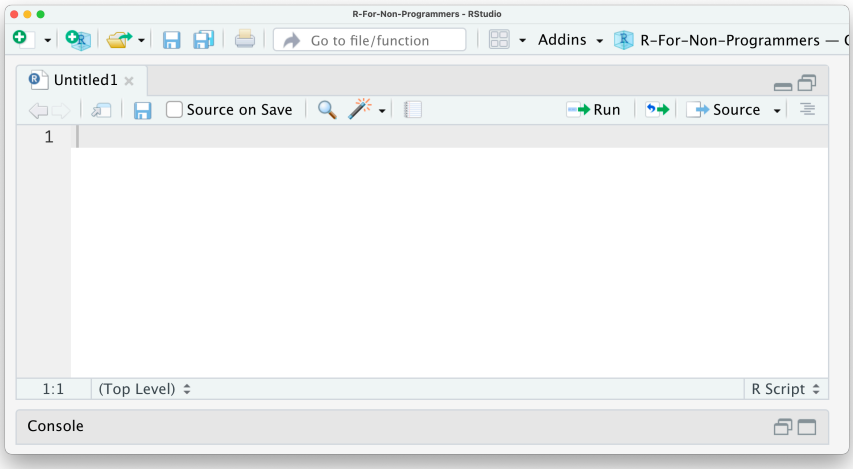
4.2 The Source window

In the top left, you can find the source window. The term 'source' can be understood as any type of file, e.g. data, programming code, notes, etc. The source panel can fulfil many functions, such as:

- Inspect data in an Excel-like format (LINK TO RELEVANT CHAPTER)
- Open programming code, e.g. an R Script (LINK TO RELEVANT CHAPTER)
- Open other text-based file formats, e.g.

  - Plain text (.txt),
  - Markdown (.md),
  - Websites (.html),
  - LaTeX (.tex),
  - BibTex (.bib),

- Edit scripts with code in it,
- Run the analysis you have written.

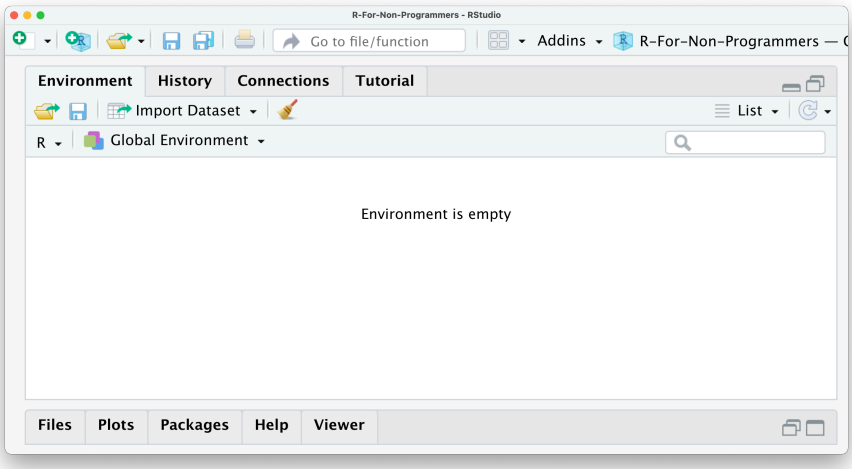


In other words, the source window will show you whatever file you are interested in, as long as RStudio can read it - and no, Microsoft Office Documents are not supported. Another limitation of the source window is that it can only show text-based files. So opening images, etc. would not work.

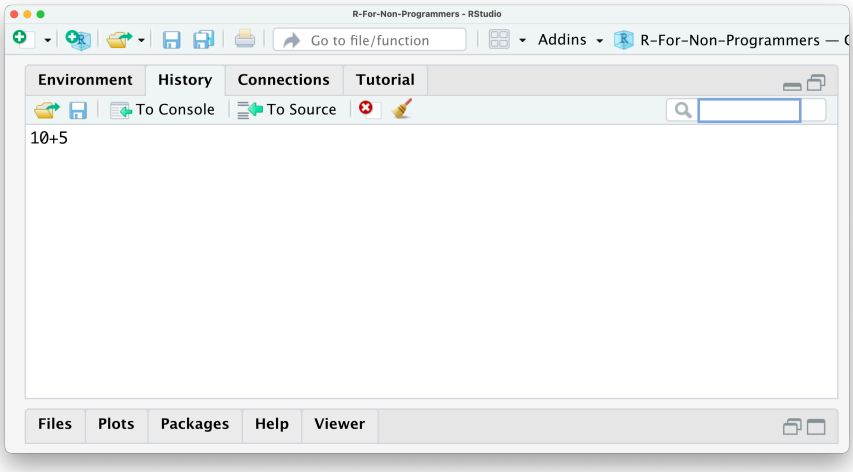
4.3 The Environment / History / Connections / Tutorial window

The window in the top right shows multiples panes. The first pane is called *Environment* and shows you objects which are available for computation. One of the first objects you will create is your dataset because, without data, we cannot perform any analysis. Thus, one object might be your data. Another object could be a plot showing the number of male and female participants in your study. To find out how to create objects yourself, you can take a glimpse at (INSERT CHAPTER X). Besides datasets and plots, you will also find other objects here, e.g. lists, vectors and functions you created yourself. Don't worry

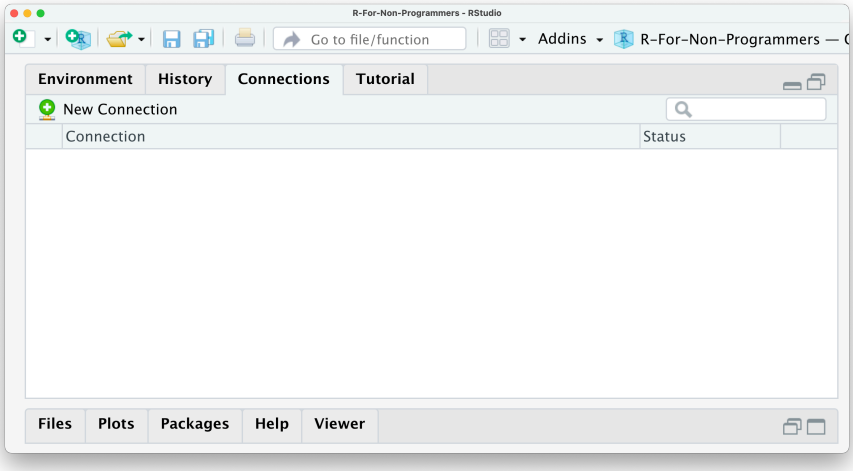
if none of these words makes sense at this point. We will cover each of them in the upcoming chapters. For now, remember this is a place where you can find different objects you created.



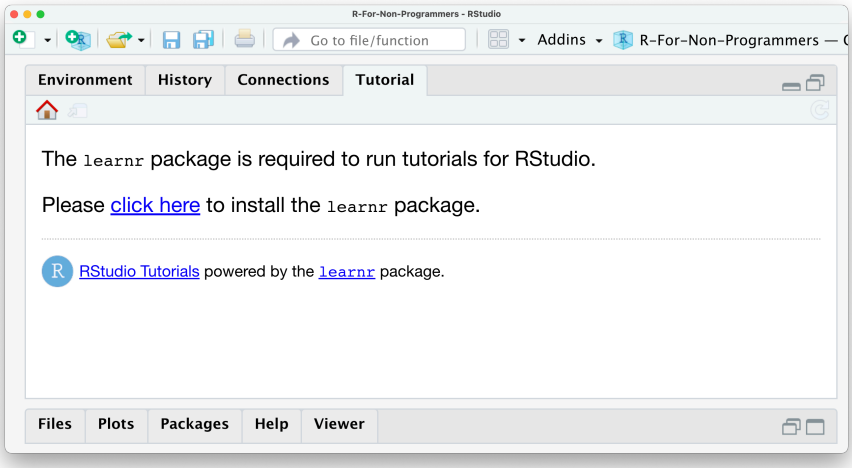
The *History* pane is very easy to understand. Whatever computation you run in the Console will be stored. So you can go back and see what you coded and rerun that code. Remember the example from above where we computed the sum of `10+5`? This computation is stored in the history of RStudio, and you can rerun it by clicking on `10+5` in the history pane and then click on `To Console`. This will insert `10+5` back into the Console, and we can hit `Return` to retrieve the result. You also have the option to copy the code into an existing or new R Script by clicking on `To Source`. By doing this, you can save this computation on your computer and reuse it later. Finally, if you would like to store your history, you can do so by clicking on the `floppy disk symbol`. There are two more buttons in this pane, one allows you to delete individual entries in the history, and the last one, a `broom`, clears the entire history (irrevocably).



The pane *Connections* allows you to tab into external databases directly. This can come in handy when you work collaboratively on the same data or want to work with extensive datasets without having to download them. However, for an introduction to R, we will not use this feature of RStudio for now.

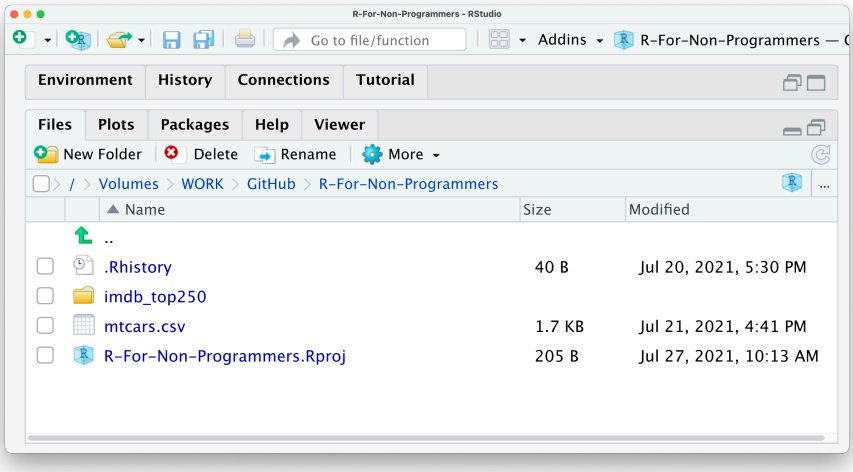


The last pane is called *Tutorial*. Here you can find additional materials to learn *R* and RStudio. If you search for more great content to learn R, this serves as a great starting point.



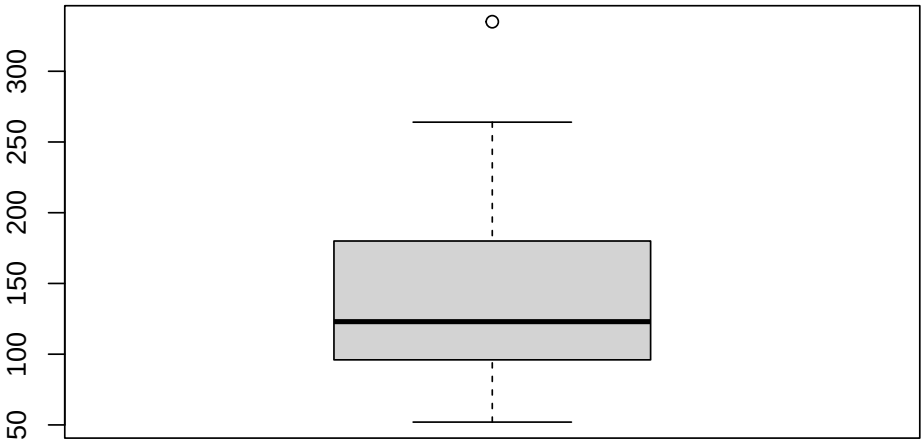
4.4 The Files / Plots / Packages / Help / Viewer window

The last window consists of five essential panes. The first one is the *Files* pane. As the name indicates, it lists all the files and folders in your root directory. A root directory is the default directory where RStudio saves your files, for example, your analysis. However, you can easily change this directory to something else (see also CHAPTER X) or use R Project files (see CHAPTER X) to carry out your research. Thus, the *Files* pane is an easy way to load data into RStudio and create folders to keep your research project well organised.



Since the Console cannot reproduce data visualisations, RStudio offers a way to do this very easily. It is through the Plots pane. This pane is exclusively designed to show you any plots you have created using R. Here is a simple example that you can try. Type into your console `boxplot(mtcars$hp)`.

```
# Here we create a nice boxplot using a dataset called 'mtcars'
boxplot(mtcars$hp)
```



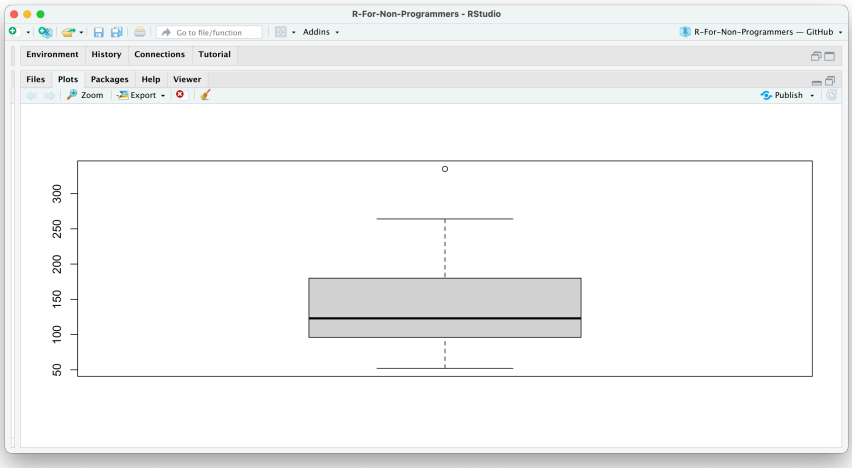
Although this is a short piece of coding, it performs quite a lot of steps:

- it uses a function called `boxplot()` to draw a boxplot of
- a variable called `hp` (for horsepower), which is located in

- a dataset named `mtcars`,

- and it renders the graph in your *Plots* pane

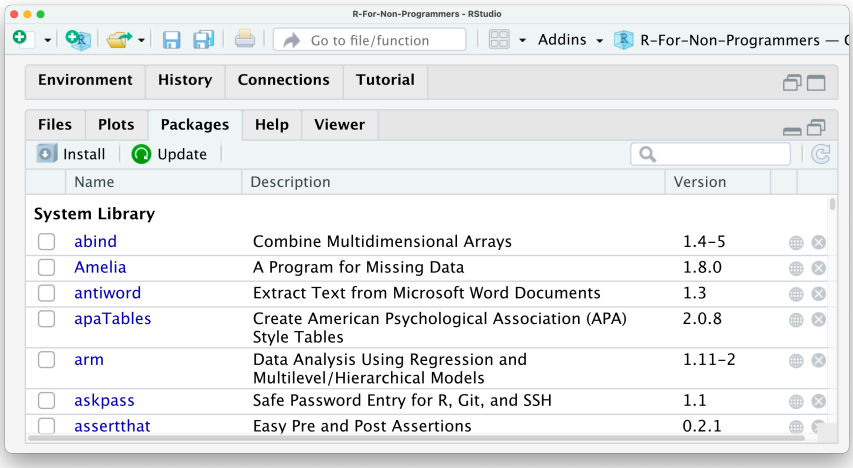
This is how the plot should look like in your RStudio *Plots* pane.



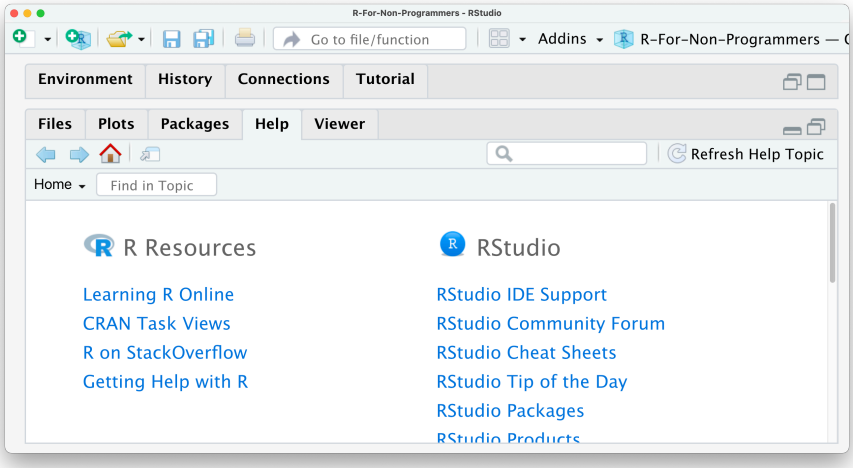
If you wish to delete the plot, you can click on the `red circle with a white x` symbol. This will delete the currently visible plot. If you wish to remove all plots from this pane, you can use the `broom`. There is also an option to export your plot and move back and forth between different plots.

Do not worry about the coding at this point. It will all make sense in the following chapters.

The next pane is called *Packages*. Packages are additional tools you can import and use when performing your analysis. A frequent analogy people use to explain packages is your phone and the apps you install. Each package you download is equivalent to an app on your phone. It can enhance different aspects of working in *R*, such as creating animated plots, using unique machine learning algorithms, or simply making your life easier by doing multiple computations with just one single line of code. You will learn more about *R packages* in Chapter 5.4.



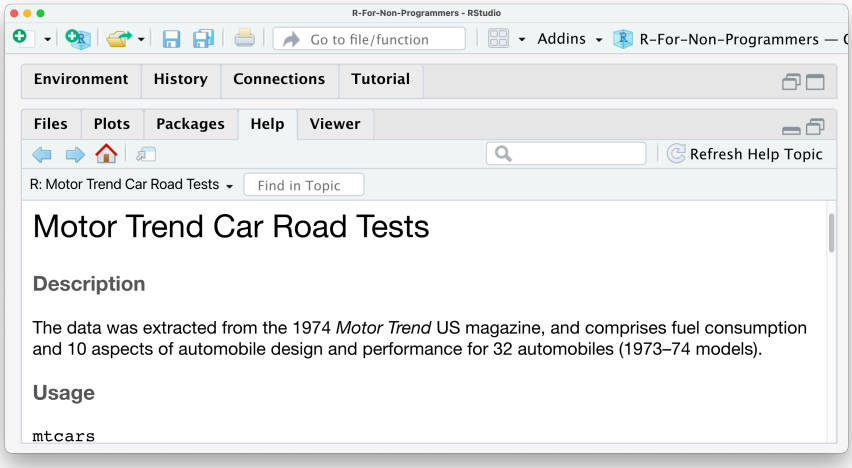
If you are in dire need of help, RStudio provides you with a *Help* pane. You can search for specific topics, for example how certain computations work. The *Help* pane also has documentation on different datasets that are included in *R*, RStudio or *R packages* you have installed. If you want a more comprehensive overview of how you can find help, have a look at CRAN's 'Getting Help with R' webpage.



So, for example, if you want to know what the `mtcars` dataset is, you can either use the search window in the *Help* pane or, much easier, use a `?` in the console to search for it:

```
# Type a '?' and immediately add the name to bring up helpful information.

?mtcars
```

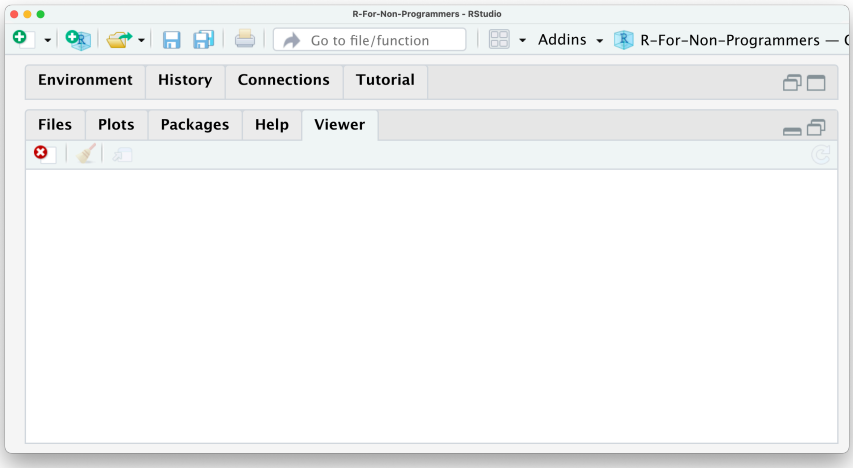
This will open the *Help* pane and give you more information about this dataset:



There are many different ways of how you can find help with your coding beyond RStudio and this book. My top three platforms to find solutions to my programming problems are:

- Google
- stackoverflow.com
- Twitter (with #RStats)

Lastly, we have the *Viewer* pane. Not every data visualisation we create in R is a static image. You can create dynamic data visualisations or even websites with R. This type of content is displayed in the Viewer pane rather than in the Plots pane. Often these visualisations are based on HTML and other web-based programming languages. As such, it is easy to open them in your browser as well. However, in this book, we mainly focus on two-dimensional static plots, which are the ones you likely need most of the time, either for your assignments, thesis, or publication.



4.5 Customise your user interface

As a last remark in this chapter, I would like to make you aware that you can modify each window. There are three basic adjustments you can make:

- Hide panes by clicking on the window symbol in the top right corner of each window,
- Resize panes by dragging the border of a window horizontally or vertically, or
- Add and remove panes by going to `RStudio > Preferences > Pane Layout`, or use the keyboard shortcut `+ ,` if you are on a Mac. There is, unfortunately no default shortcut for PC users.

If you want a fully customised experience you can also alter the colour scheme of the RStudio itself (`RStudio > Preferences > Appearance`) and if the themes offered are not enough for you, you can create a custom theme here

Chapter 5

R Basics: The very fundamentals

After a likely tedious installation of R and RStudio, as well as a somewhat detailed introduction to the RStudio interface, you are finally ready to 'do' things. By 'doing', I mean coding. The term 'coding' in itself can instil fear in some of you, but you only need one skill to do it: Writing. As mentioned earlier, learning coding or programming means learning a new language. However, once you have the basic grammar down, you already can communicate quite a bit. In this section, we will explore the fundamentals of R. These build the foundation for everything that follows. After that, we dive right into some analysis.

5.1 Basic computations in R

The most basic computation you can do in R is arithmetic operations. In other words, addition, subtraction, multiplication, division, exponentiation and extraction of roots. In other words, R can be used like your pocket calculator, or more likely the one you have on your phone. For example, in Chapter 4.1 we already performed an addition. Thus, it might not come as a surprise how their equivalents work in R. Let's take a look at the following examples:

```
# Addition
10 + 5
## [1] 15

# Subtraction
10 - 5
## [1] 5
```

```
# Multiplication
10 * 5
## [1] 50

# Division
10 / 5
## [1] 2

# Exponentiation
10 ^ 2
## [1] 100

# Square root
sqrt(10)
## [1] 3.162278
```

They all look fairly straightforward except for the extraction of roots. As you probably know, extracting the root would typically mean we use the symbol $\sqrt{}$ on your calculator. To compute the square root in R, we have to use a function instead to perform the computation. So we first put the name of the function `sqrt` and then the value `10` within parenthesis `()`. This results in the following code: `sqrt(10)`. If we were to write this down in our report, we would write $\sqrt[2]{10}$.

Functions are an essential part of R and programming in general. You will learn more about them in this chapter.Besides arithmetic operations, there are also logical queries you can perform. Logical queries always return either the value TRUE or FALSE. Here are some examples which make this clearer:

```
#1 Is it TRUE or FALSE?
1 == 1
## [1] TRUE

#2 Is 45 bigger than 55?
45 > 55
## [1] FALSE

#3 Is 1982 bigger or equal to 1982?
1982 >= 1982
## [1] TRUE

#4 Are these two words NOT the same?
"Friends" != "friends"
## [1] TRUE
```

```
#5 Are these sentences the same?
"I love statistics" == "I love statistícs"
## [1] FALSE
```

Reflecting on these examples, you might notice three important things:

1. I used `==` instead of `=`,
2. I can compare non-numerical values, i.e. text, which is also known as `character` values, with each other,
3. The devil is in the details (considering #5).

One of the most common mistakes of R novices is the confusion around the `==` and `=` notation. While `==` represents `equal to`, `=` is used to assign a value to an object (for more details on assignments see Chapter 4.4). However, in practice, most R programmers tend to avoid `=` since it can easily lead to confusion with `==`. As such, you can strike this one out of your R vocabulary for now.

There are many different logical operations you can perform. Table 5.1 lists the most frequently used logical operators for your reference. These will become important once we select only certain parts of our data for analysis, e.g. only `female` participants.

Table 5.1: Logical Operators in R

Operator	Description
==	is equal to
>=	is bigger or equal to
<=	is smaller of equal to
!=	is not equal to
a | b	a or b
a & b	a and b
!a	is not a

5.2 Assigning values to objects: '<-'

Another common task you will perform is assigning values to an object. An object can be many different things:

- a dataset,
- the results of a computation,
- a plot,

- a series of numbers,
- a list of names,
- a function,
- etc.

In short, an object is an umbrella term for many different things which form part of your data analysis. For example, objects are handy when storing results that you want to process further in later analytical steps. Let's have a look at an example.

```
# I have a friend called "Fiona"
friends <- "Fiona"
```

In this example, I created an object called `friends` and added `"Fiona"` to it. Remember, because `"Fiona"` represents a `string`, we need `""`. So, if you wanted to read this line of code, you would say, '`friends` gets the value `"Fiona"`'. Alternatively, you could also say '`"Fiona"` is assigned to `friends`'.

If you look into your environment pane, you will find the object we just created. You can see it carries the value `"Fiona"`. We can also print values of an object in the console by simply typing the name of the object `friends` and hit `Return` .

```
# Who are my friends?
friends
## [1] "Fiona"
```

Sadly, it seems I only have one friend. Luckily we can add some more, not the least to make me feel less lonely. To create objects with multiple values, we can use the function `c()`, which stands for 'concatenate'. The Cambridge Dictionary (2021) define this word as follows:

> '***concatenate***',
>
> to put things together as a connected series

Let's concatenate some more friends into our `friends` object.

```
# Adding some more friends to my life
friends <- c("Fiona", "Ida", "Lukas", "Georg", "Daniel", "Pavel", "Tigger")

# Here are all my friends
friends
## [1] "Fiona"  "Ida"    "Lukas"  "Georg"  "Daniel" "Pavel"  "Tigger"
```

To concatenate values into a single object, we need to use a comma `,` to separate each value. Otherwise, R will report an error back.

```
friends <- c("Fiona" "Ida")
## Error: <text>:1:22: unexpected string constant
## 1: friends <- c("Fiona" "Ida"
##                          ^
```

R's error messages tend to be very useful and give meaningful clues to what went wrong. In this case, we can see that something 'unexpected' happen, and it shows where our mistake is.

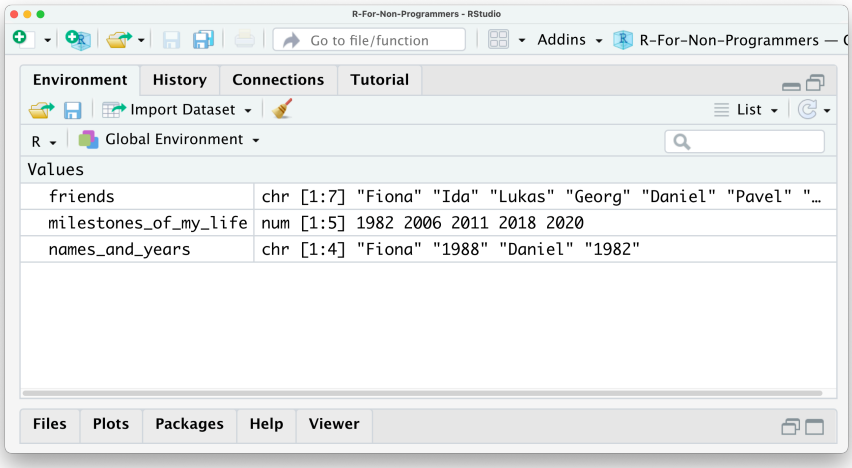
You can also concatenate numbers, and if you add `()` around it, you can automatically print the content of the object to the console. Thus, `(milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020))` is the same as `milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020)` followed by `milestones_of_my_life`. The following examples illustrate this.

```
# Important years in my life
milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020)
milestones_of_my_life
## [1] 1982 2006 2011 2018 2020

# The same as above, but we don't need the second line of code
(milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020))
## [1] 1982 2006 2011 2018 2020
```

Finally, we can also concatenate numbers and character values into one object:

```
(names_and_years <- c("Fiona", 1988, "Daniel", 1982))
## [1] "Fiona"  "1988"   "Daniel" "1982"
```

This last example is not necessarily something I would recommend to do, because it likely leads to undesirable outcomes. If you look into your environment pane you currently have three objects: `friends`, `milestones_of_my_life`, and `names_and_years`.



The `friends` object shows that all the values inside the object are classified as `chr`, which denominates `character`. In this case, this is correct because it only includes the names of my friends. On the other hand, the object `milestones_of_my_life` only includes `numeric` values, and therefore it says `num` in the environment pane. However, for the object `names_and_years` we know we want to have `numeric` and `character` values included. Still, R recognises them as `character` values only because values inside objects are meant to be of the same type.

Consequently, mixing different types of data (as explained in Chapter @ref()) into one object is likely a bad idea. This is especially true if you want to use the numeric values for computation. In short: ensure your objects are all of the same data type.

There is an exception to this rule. 'Of course', you might say. There is one object that can have values of different types: `list`. As the name indicates, a `list` object holds several items. These items are usually other objects. In the spirit of 'Inception', you can have lists inside lists, which contain more objects.

Let's create a list called `x_files` using the `list` function and place all our objects inside.

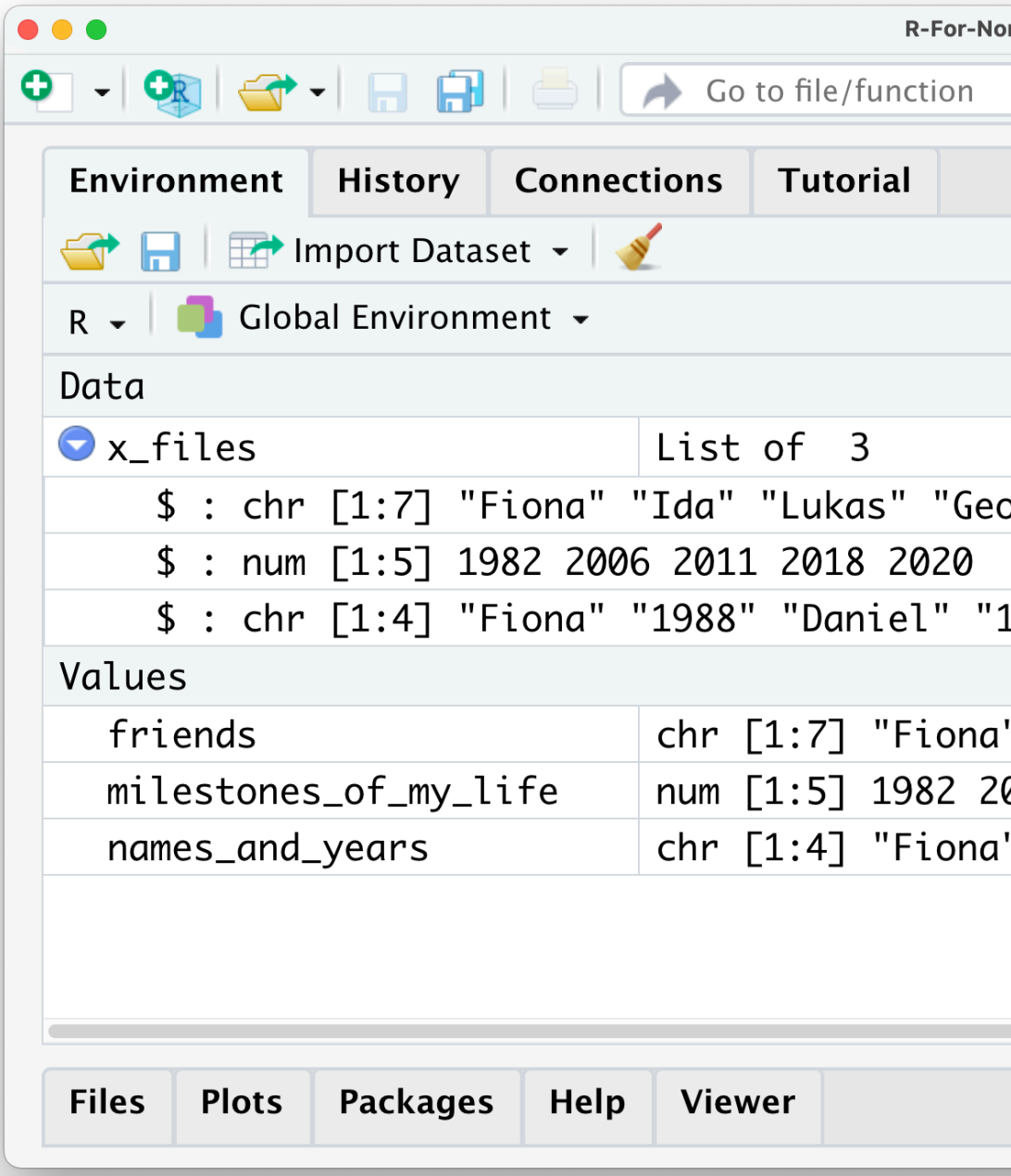
```
# This creates our list of objects
x_files <- list(friends,
                milestones_of_my_life,
                names_and_years)

# Let's have a look what is hidden inside the x_files
x_files
## [[1]]
```

```
## [1] "Fiona"  "Ida"    "Lukas"  "Georg"  "Daniel" "Pavel"  "Tigger"
##
## [[2]]
## [1] 1982 2006 2011 2018 2020
##
## [[3]]
## [1] "Fiona"  "1988"   "Daniel" "1982"
```

You will notice in this example that I do not use `""` for each value in the list. This is because `friends` is not a character I put into the list, but an object. When we refer to objects, we do not need quotation marks.

We will encounter `list` objects quite frequently when we perform our analysis. Some functions return the results in the format of lists. This can be very helpful because otherwise our environment pane will be littered with objects. We would not necessarily know how they relate to each other, or worse, to which analysis they belong. Looking at the list item in the environment page (Figure 5.2), you can see that the object `x_files` is classified as a `List of 3,` and if you click on the blue icon, you can inspect the different objects inside.

\begin{figure}


R-For-No
Go to file/function
Environment
History
Connections
Tutorial
Import Dataset
R
Global Environment
Data
x_files
List of 3
$: chr [1:7] "Fiona" "Ida" "Lukas" "Geo
$: num [1:5] 1982 2006 2011 2018 2020
$: chr [1:4] "Fiona" "1988" "Daniel" "1
Values
friends
chr [1:7] "Fiona'
milestones_of_my_life
num [1:5] 1982 20
names_and_years
chr [1:4] "Fiona'
Files
Plots
Packages
Help
Viewer

{

}

\caption{The environment pane showing our objects and our list `x_files`}
\end{figure}

In Chapter 5.1, I mentioned that we should avoid using the `=` operator and explained that it is used to assign values to objects. You can, if you want, use `=` instead of `<-`. They fulfil the same purpose. However, as mentioned before, it is not wise to do so. Here is an example that shows that, in principle, it is possible.

```
# DO
(avengers1 <- c("Iron Man", "Captain America", "Black Widow", "Vision"))
## [1] "Iron Man"        "Captain America" "Black Widow"     "Vision"

# DON'T
(avengers2 = c("Iron Man", "Captain America", "Black Widow", "Vision"))
## [1] "Iron Man"        "Captain America" "Black Widow"     "Vision"
```

On a final note, naming your objects is limited. You cannot chose any name. First, every name needs to start with a letter. Second, you can only use letters, numbers `_` and `.` as valid components of the names for your objects (see also Wickham and Grolemund, 2016, Chapter 4.2.). I recommend to establish a naming convention that you adhere to. Personally I prefer to only user lower letters and `_` to separate/connect words. You want to keep names informative, succinct and precise. Here are some examples of what some might consider good and bad choices for names.

```
# Good choices
income_per_annum
open_to_exp          # for 'openness to new experiences'
soc_int              # for 'social integration'

# Bad choices
IncomePerAnnum
measurement_of_boredom_of_watching_youtube
Sleep.per_monthsIn.hours
```

Ultimately, you need to be able to effectively work with your data and output. Ideally, this should be true for others as well who want or need to work with your R project as well, e.g. your co-investigator or supervisor. The same is true for your column names in datasets (see Chapter @ref()). Some more information about coding style (i.e. the style of writing coding) can be found in Chapter 5.5.

5.3 Functions

I used the term 'function' multiple times, but I never thoroughly explained what they are and why we need them. In simple terms, functions are objects. They contain lines of code that someone has written for us or we have written ourselves. One could say they are code snippets ready to use. Someone else might see them as shortcuts for our programming. Functions increase the speed with which we perform our analysis and write our computations and make our code more readable. Consider computing the `mean` of values stored in the object `pocket_money`.

```
# First we create an object that stores our desired values
pocket_money <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89)

#1 Manually compute the mean
sum <- 0 + 1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 + 34 + 55 + 89
sum / 12 # There are 12 items in the object
## [1] 19.33333

#2 Use a function to compute the mean
mean(pocket_money)
## [1] 19.33333

#3 Let's make sure #1 and #2 are actually the same
sum / 12 == mean(pocket_money)
## [1] TRUE
```

If we manually compute the mean, we first calculate the sum of all values in the object `pocket_money`[1]. Then we divide it by the number of values in the object, which is `12`. This is the traditional way of computing the mean as we know it from primary school. However, by simply using the function `mean()`, we not only write considerably less code, but it is also much easier to understand as well because the word `mean` does precisely what we would expect. Which one do you find easier?

To further illustrate how functions look like, let's create one ourselves and call it `my_mean`.

```
my_mean <- function(numbers){
  sum <- sum(numbers)               # Compute the sum of all values in 'numbers'
  result <- sum/length(numbers)     # Divide the sum by the number of items in 'numbers
  return(result)                    # Return the result in the console
}
```

[1] If you find the order of numbers suspicious, it is because it represents the famous Fibonacci sequence.

```
my_mean(pocket_money)
## [1] 19.33333
```

Do not worry if half of this code does not make sense to you. Writing functions is an advanced R skill. However, it is good to know how functions look on the 'inside'. You certainly can see the similarities between the code we have written before, but instead of using actual numbers, we work with placeholders like `numbers`. This way, we can use a function for different data and do not have to rewrite it every time.

All functions in R share the same structure. They have a `name` followed by `()`. Within these parentheses, we put `arguments`, which have specific `values`. For example, a function would look something like this:

```
name_of_function(argument_1 = value_1,
                 argument_2 = value_2,
                 argument_3 = value_3)
```

How many arguments there are and what kind of values you can provide is very much dependent on the function you use. Thus, not every function takes every value. In the case of `mean()`, the function takes an object which holds a sequence of `numeric` values. It would make very little sense to compute the mean of our `friends` object, because it only contains names. R would return an error message:

```
mean(friends)
## Warning in mean.default(friends): argument is not numeric or logical: returning
## NA
## [1] NA
```

`NA` refers to a value that is *'not available'*. In this case, R tries to compute the mean, but the result is not available, because the values are not `numeric` but a `character`. In your dataset, you might find cells that are `NA`, which means there is data missing. Remember: If a function attempts a computation that includes even just a single value that is `NA`, R will return `NA`. However, there is a way to fix this. You will learn more about how to deal with `NA` values in Chapter @ref().

Sometimes you will also get a message from R that states `NaN`. `NaN` stands for *'not a number'* and is returned when something is not possible to compute, for example:

```
# Example 1
0/0
## [1] NaN
```

```
# Example 2
sqrt(-9)
## Warning in sqrt(-9): NaNs produced
## [1] NaN
```

5.4 R packages

R has many built-in functions that we can use right away. However, some of the most interesting ones are developed by different programmers, data scientists and enthusiasts. To add more functions to your repertoire, you can install R packages. R packages are a collection of functions that you can download and use for your own analysis. Throughout this book, you will learn about and use many different R packages to accomplish various tasks.

To give you another analogy,

- R is like a global supermarket,
- RStudio is like my shopping cart,
- and R packages are the products I can pick from the shelves.

Luckily, R packages are free to use, so I do not have to bring my credit card. For me, these additional functions, developed by some of the most outstanding scientists, is what keeps me addicted to performing my research in R.

R packages do not only include functions but often include datasets and documentation of what each function does. This way, you can easily try every function right away, even without your own dataset and read through what each function in the package does. Figure 5.1

However, how do you find those R packages? They are right at your fingertips. You have two options:

1. Use the function `install.packages()`
2. Use the packages pane in RStudio (see Chapter 4.4)

5.4.1 Installing packages using `install.packages()`

The simplest and fastest way to install a package is calling the function `install.packages()`. You can either use it to install a single package or install a series of packages all at once using our trusty `c()` function. All you need to know is the name of the package. This approach works for all packages that are on CRAN (remember CRAN from Chapter 3.1?).

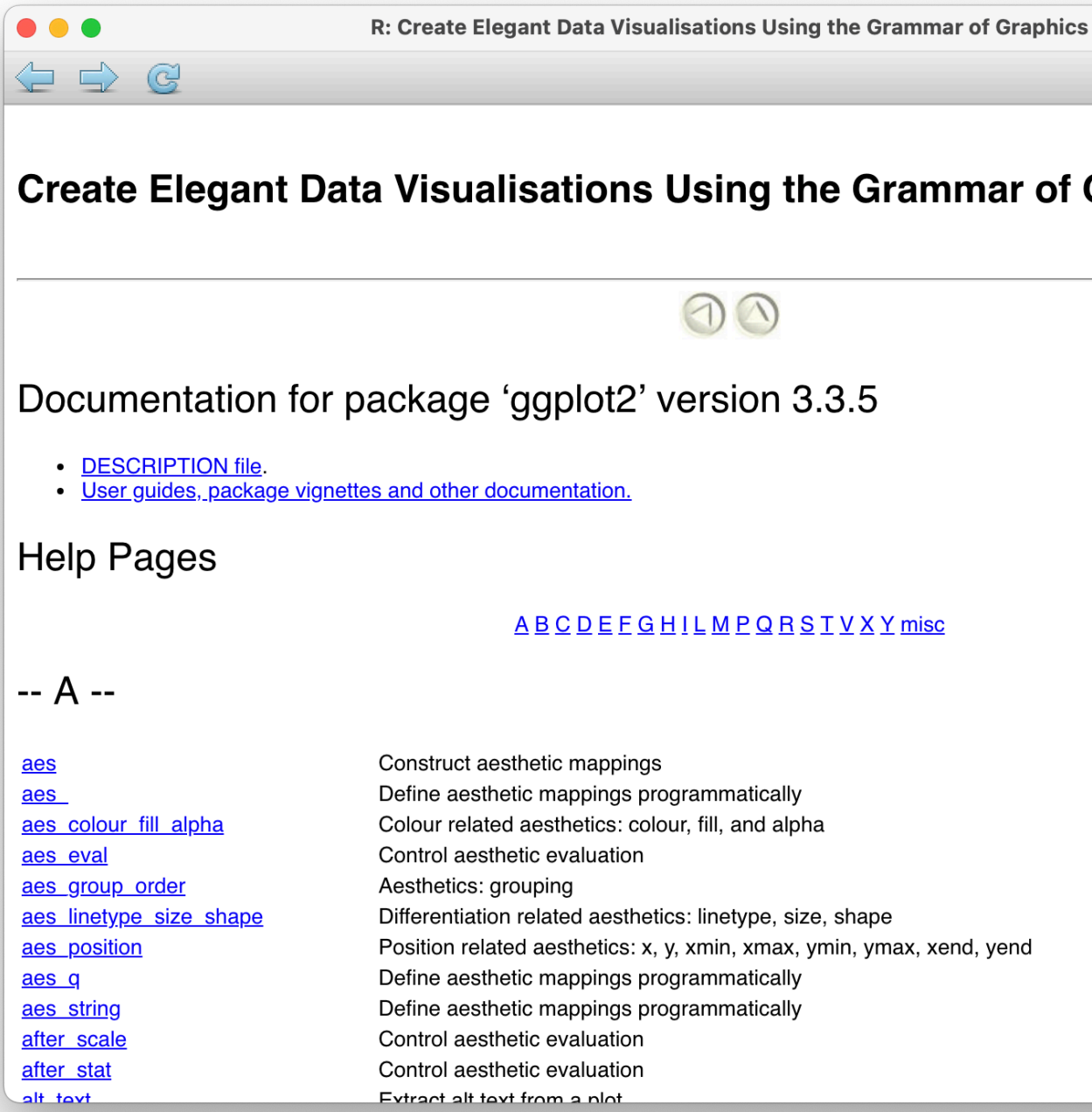


Figure 5.1: The R package documentation for 'ggplot2'

```
# Install a single package
install.packages("tidyverse")

# Install multiple packages at once
install.packages(c("tidyverse", "naniar", "psych"))
```

If a package is not available from CRAN, chances are you can find them on GitHub. GitHub is probably the world's largest global platform for programmers from all walks of life, and many of them develop fantastic R packages that make R programming not just easier but a lot more fun. As you continue to work in R, you should seriously consider creating your own account to keep backups of your R projects (see also Chapter 13.1).

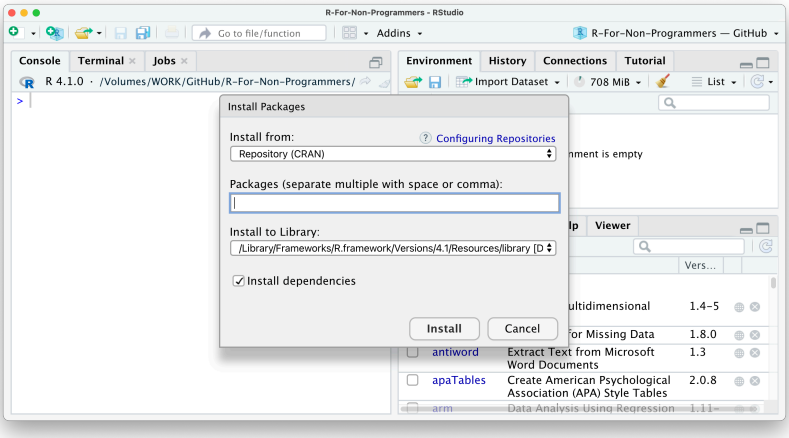
An essential companion for this book is `r4np`, which contains all datasets for this book and some useful functions to get you up and running in no time.

```
# Install the 'r4np' pacakge from GitHub
devtools::install_github("ddauber/r4np")
```

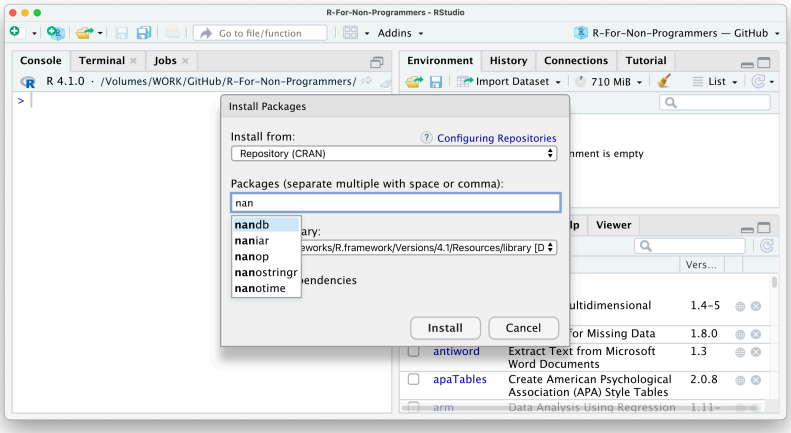
5.4.2 Installing packages via RStudio's package pane

RStudio offers a very convenient way of installing packages. In the packages pane, you cannot only see your installed packages, but you have two more buttons: `Install` and `Update`. The names are very self-explanatory. To install an R package you can follow the following steps:

1. Click on `Install`.
2. In most cases, you want to make sure you have `Repository (CRAN)` selected.

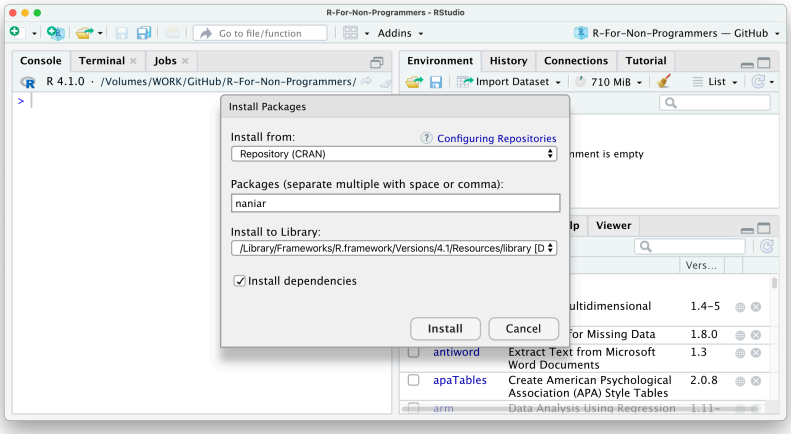


3. Type in the name of the package you wish to install. RStudio offers an auto-complete feature to make it even easier to find the package you want.



4. I recommend NOT to change the option which says `Install to library`. The default library settings will suffice.

5. Finally, I recommend to select `Install dependencies`, because some packages need other packages to function properly. This way, you do not have to do this manually.



The only real downside of using the packages pane is that you cannot install packages hosted on GitHub only. However, you can download them from there and install them directly from your computer using this option. This is particularly useful if you do not have an internet connection but you already downloaded the required packages onto a hard drive.

5.4.3 Using R Packages

Now that you have a nice collection of R packages, the next step would be to use them. While you only have to install R packages once, you have to 'activate' them every time you start an new session in RStudio. This process is also called 'loading an R package'. Once an R package is loaded, you can use all its functions. To load an R package, we have to use the function `library()`.

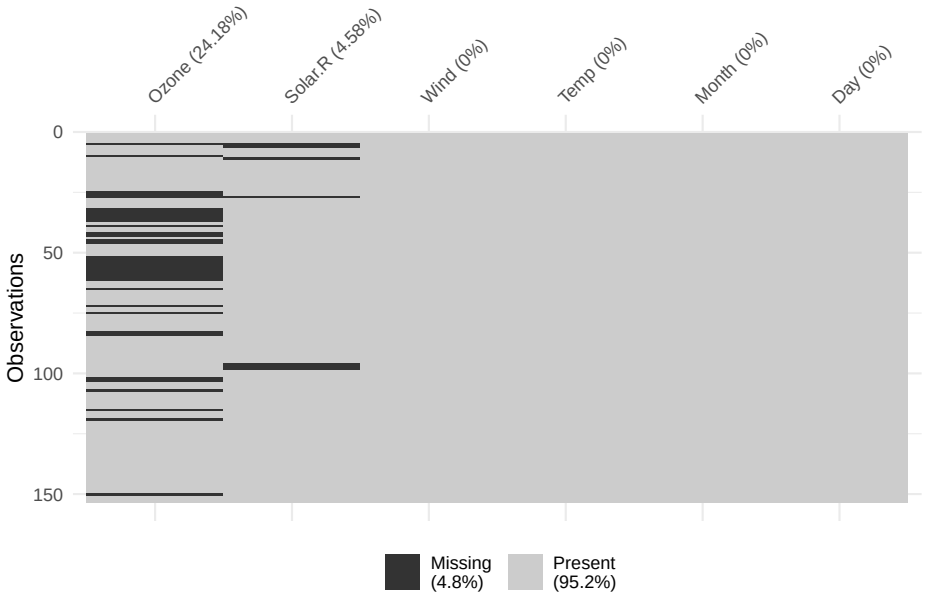
```
library(tidyverse)
```

The `tidyverse` package is a special kind of package. It contains multiple packages and loads them all at once. Almost all included packages (and more) you will use at some point when working through this book.

I know what you are thinking. Can you use `c()` to load all your packages at once? Unfortunately not. However, there is a way to do this, but it goes beyond the scope of this book to fully explain this (if you are curious, you can take a peek here).

Besides, it is not always advisable to load all functions of an entire package. One reason could be that two packages contain a function with the same name but with a different purpose. Two functions with the same name create a conflict between these two packages, and one of the functions would not be usable. Another reason could be that you only need to use the function once, and loading the whole package to use only one specific function seems excessive. Instead, you can explicitly call functions from packages without loading the package. For example, we might want to use the `vismis()` function from the `naniar` package to show where data is missing in our dataset `airquality`. Writing the code this way is also much quicker than loading the package and then calling the function if you don't use it repeatedly. Copy the code and try it yourself. Make sure you have `naniar` installed (see above). We will work with this package when we explore missing data in Chapter @ref().

```
# Here I use the dataset 'airquality', which comes with R
naniar::vis_miss(airquality)
```



5.5 Coding etiquette

Now you know everything to get started, but before we jump into our first project, I would like to briefly touch upon coding etiquette. This is not something that improves your analytical or coding skills directly, but is essential in building good habbits and making your life and those of others a little easier. Consider writing code like growing plants in your garden. You want to nurture the good plants, remove the weed and add labels that tell you which plant it is that you are growing. At the end of the day, you want your garden to be well-maintained. Treat you programming code the same way.

A script (see Chapter @ref()) with code should always have at least the following qualities:

- Only contains code that is necessary,
- Is easy to read and understand,
- Is self-contained.

With simple code this is easily achieved. However, what about more complex and longer code representing a whole set of analytical steps?

```
# Very messy code

library(tidyverse)
library(jtools)
model1 <- lm(covid_cases_per_1m ~ idv, data = df)
summ(model1, scale = TRUE, transform.response = TRUE, vifs = TRUE)
df %>% ggplot(aes(covid_cases_per_1m, idv, colour = europe, label = country))+
theme_minimal()+ geom_label(nudge_y = 2) + geom_point()
mod_model2 <- lm(cases_per_1m ~ idv + uai + idv*europe + uai*europe, data = df)
summ(mod_model2, scale = TRUE, transform.response = TRUE, vifs = TRUE)
anova(mod_model1, mod_model2)
```

How about the following in comparison?

```
# Nicely structured code

# Load required R packages
library(tidyverse)
library(jtools)

# ---- Modelling COVID-19 cases ----

## Specify and run a regression
model1 <- lm(covid_cases_per_1m ~ idv, data = df)

## Retrieve the summary statistics of model1
summ(model1, scale = TRUE, transform.response = TRUE, vifs = TRUE)

# Does is matter whether a country lies in Europe?

## Visualise rel. of covid cases, idv and being a European country
df %>%
  ggplot(aes(covid_cases_per_1m, idv, colour = europe, label = country))+
  theme_minimal()+
  geom_label(nudge_y = 2) +
  geom_point()

## Specify and run a revised regression
mod_model2 <- lm(cases_per_1m ~ idv + uai + idv*europe + uai*europe, data = df)

## Retrieve the summary statistics of model2
summ(mod_model2, scale = TRUE, transform.response = TRUE, vifs = TRUE)

## Test whether model2 is an improvement over model1
anova(mod_model1, mod_model2)
```

I hope we can agree that the second example is much easier to read and understand even though you probably do not understand most of it yet. For once, I separated the different analytical steps from each other like paragraphs in a report. Apart from that, I added comments with `#` to provide more context to my code for someone else who wants to understand my analysis. Admittedly, this example is a little excessive. Usually, you might have fewer comments. Commenting is an integral part of programming because it allows you to remember what you did. Ideally, you want to strike a good balance between commenting on and writing your code. How many comments you need will likely change throughout your R programming journey. Think of comments as headers for your programming script that give it structure..

We can use `#` not only to write comments but also to tell R not to run particular code. This is very helpful if you want to keep some code but do not want to use it yet. There is also a handy keyboard shortcut you can use to 'deactivate' multiple lines of code at once. Select whatever you want to 'comment out' in your script and press `Ctrl+Shift+C` (PC) or `Cmd+Shift+C` (Mac).

```
# mean(pocket_money) # R will NOT run this code
mean(pocket_money)   # R will run this code
```

RStudio helps a lot with keeping your coding tidy and properly formatted. However, there are some additional aspects worth considering. If you want to find out more about coding style, I highly recommend to read through the *'The tidyverse style guide'* (Wickham, Hadley, 2021).

5.6 Exercises

1. What is the result of $\sqrt[2]{25-16}+2*8-6$?
2. What does the console return if you execute the following code `"Five" == 5`?
3. Create a list called `books` and include the following book titles in it:
    - "Harry Potter and the Deathly Hallows",
    - "The Alchemist",
    - "The Davinci Code",
    - "R For Dummies"
4. Copy and paste the function below into your RStudio console and run it. What does the function do when you use it?

```r
x_x <- function(number1, number2){
  result1 <- number1*number2
  result2 <- sqrt(number1)
  result3 <- number1-number2
  return(c(result1, result2, result3))
}
```

5. What are the three steps to use a new R package that you found on CRAN?

Check you answers: Solutions 14.1

Chapter 6

Starting your R projects

Every project likely fills you with enthusiasm and excitement. And it should. You are about to find answers to your questions, and you hopefully come out more knowledgeable due to it. However, there are likely certain aspects of data analysis that you find less enjoyable. I can think of two:

- Keeping track of all the files my project generates
- Data wrangling

While we cover data wrangling in great detail later (Chapter 7), I would like to share some insights from my work that helped me stay organised and, consequently, less frustrated. The following applies to small and large research projects, which makes it very convenient no matter the situation. Of course, feel free to tweak my approach to whatever suits you. However, consistency is king.

6.1 Creating an R Project file

When working on a project, you likely create many different files for various purposes, especially R Scripts (see Chapter 6.3). If you are not careful, this file is stored in your system's default location, which might not be where you want them to be. RStudio allows you to manage your entire project intuitively and conveniently through R Project files. Using R Project files comes with a couple of perks, for example:

- All the files that you generate are in the same place. Your data, your coding, your exported plots, your reports, etc., all are in one place together without you having to manage the files manually.

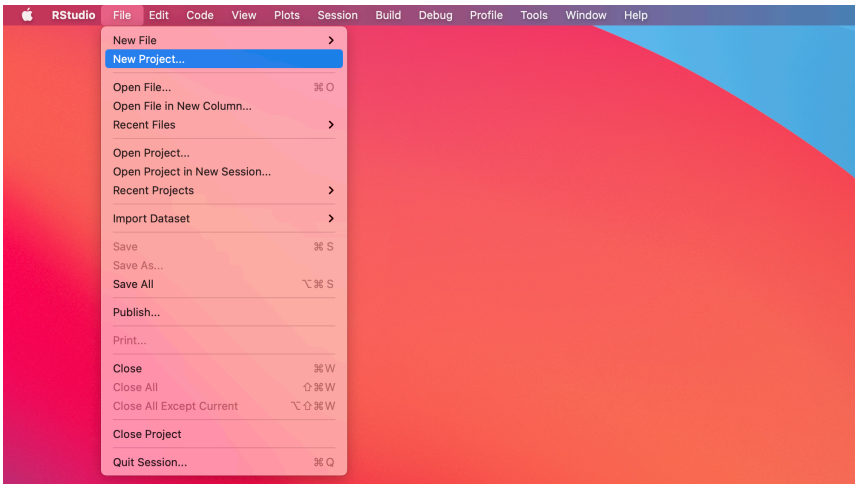
- If you want to share your project, you can share the entire folder, and others can quickly reproduce your research or help fix problems. This is because all file paths are relative and not absolute.

- You can, more easily, use GitHub for backups and so-called 'version control', which allows you to track changes you have made to your code over time (see also Chapter 13.1).

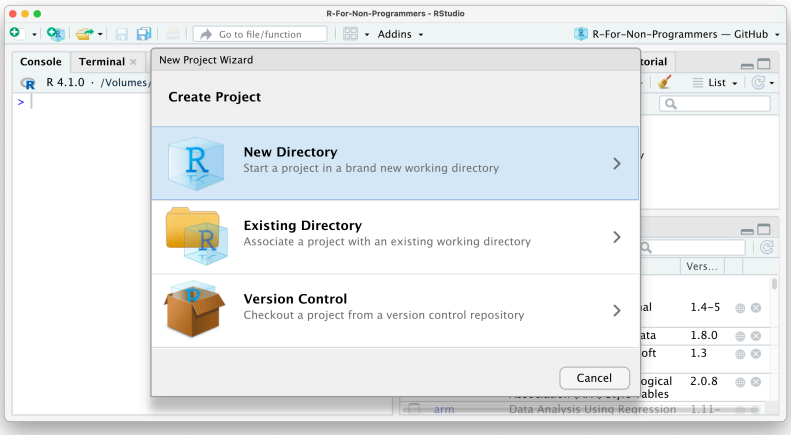
For now, the most important reason to use R Project files is the convenience of the organisation of files and the ability to share it easily with co-investigators, your supervisor, or your students.

To create an R Project, you need to perform the following steps:

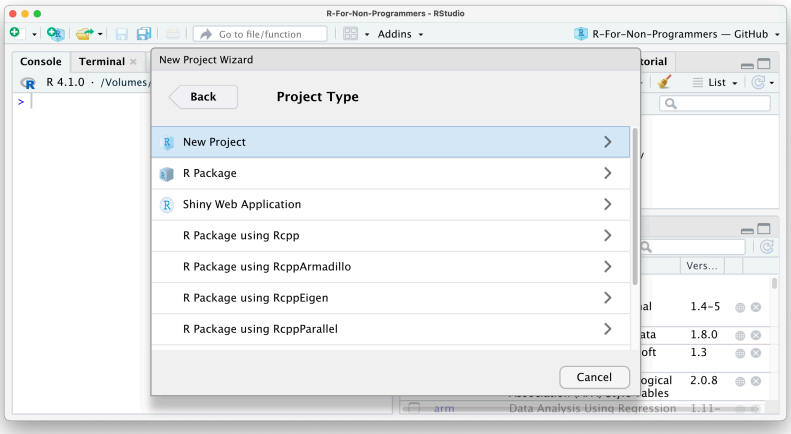
1. Select `File > New Project…` from the menu bar.



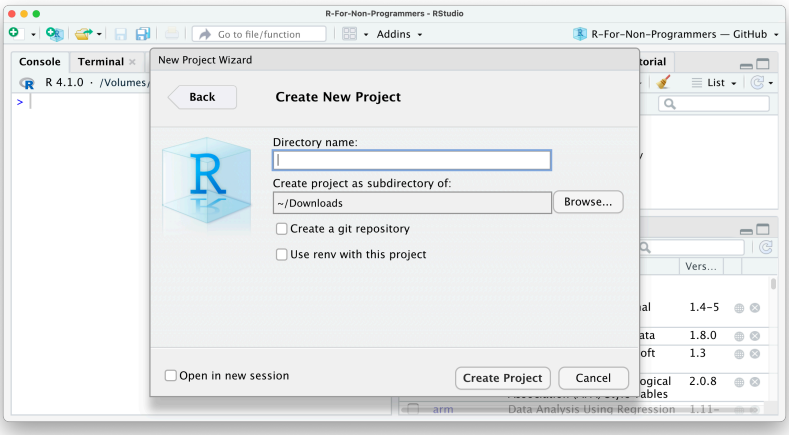
2. Select `New Directory` from the popup window.



3. Next, select `New Project`.

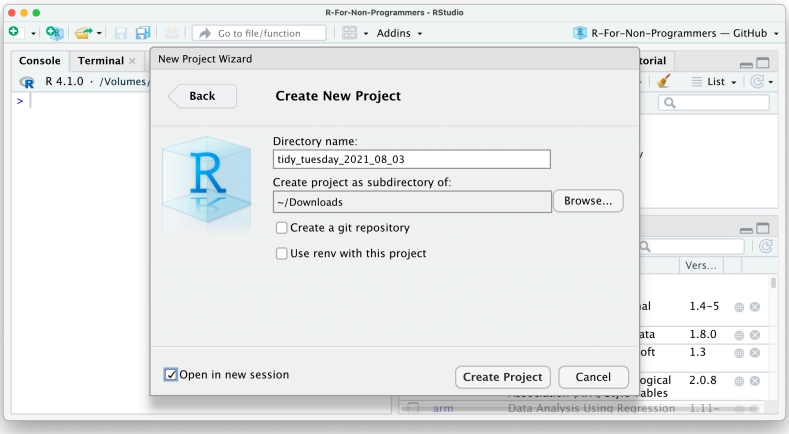


4. Pick a meaningful name for your project folder, i.e. the `Directory Name`. Ensure this project folder is created in the right place. You can change the `subdirectory` by clicking on `Browse...`. Ideally the subdirectory is a place where you usually store your research projects.

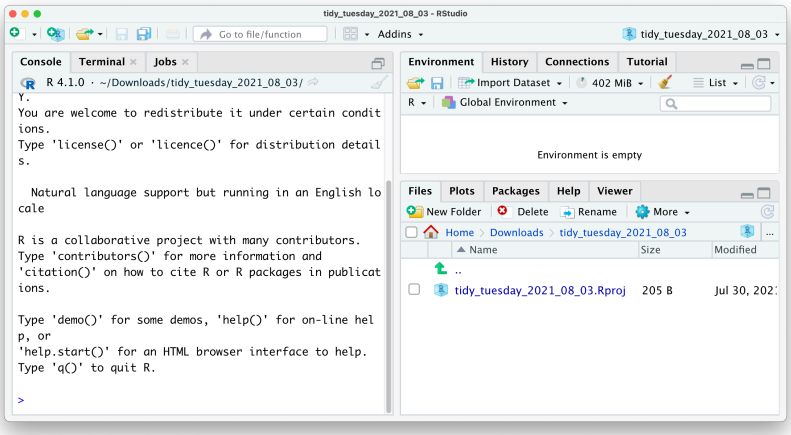


5. You have the option to `Create a git repository`. This is only relevant if you already have a GitHub account and wish to use version control. For now, you can happily ignore it.

6. Lastly, tick `Open in new session`. This will open your R Project in a new RStudio window.



7. Once you are happy with your choices, you can click `Create Project`. This will open a new R Session, and you can start working on your project.



If you look carefully, you can see that your RStudio is now 'branded' with your project name. At the top of the window, you see the project name, the files pane shows the root directory where all your files will be, and even the console shows on top the file path of your project. You could set all this up manually, but I would not recommend it, not the least because it is easy to work with R Projects.

6.2 Organising your projects

This section is not directly related to RStudio, R or data analysis in general. Instead, I want to convey to you that a good folder structure can go a long way. It is an excellent habit to start thinking about folder structures before you start working on your project. Placing your files into dedicated folders, rather than keeping them loosely in one container, will speed up your work and save you from the frustration of not finding the files you need. I have a template that I use regularly. You can either create it from scratch in RStudio or open your file browser and create the folders there. RStudio does not mind which way you do it. If you want to spend less time setting this up, you might want to use the function `create_dr()` from the `r4np` package. It creates all the folders as shown in Figure 6.1.

```
# Install 'r4np' from GitHub
devtools::install_github("ddauber/r4np")

# Create the template structure
r4np::create_dr()
```

To create a folder, click on `New Folder` in the Files pane. I usually have at least the following folders for every project I am involved in:

- A folder for my raw data. I store 'untouched' datasets in it. With 'untouched', I mean they have not been processed in any way and are usually files I downloaded from my data collection tool, e.g. online questionnaire platform.
- A folder with 'tidy' data. This is usually data I exported from R after cleaning it, i.e. after data wrangling (see Chapter 7).
- A folder for my R scripts
- A folder for my plots
- A folder for reports

Thus, in RStudio, it would look something like this:

You probably noticed that my folders have numbers in front of them. I do this to ensure that all folders are in the order I want them to be, usually not the alphabetical order my computer suggests. I use two digits because I may have more than nine folders for a project, and folder ten would otherwise be listed as the third folder in this list. With this filing strategy in place, it will be easy to find whatever I need. Even others can easily understand what I stored where. It is simply 'tidy', similar to how we want our data to be.

6.3 Creating an R Script

Code quickly becomes long and complex. Thus, it is not very convenient to write it in the console. So, instead, we can write code into an R Script. An R Script is a document that RStudio recognises as R programming code. Files that are not R Scripts, like `.txt`, `.rtf` or `.md`, can also be opened in RStudio, but any code written in it will not be automatically recognised.

When opening an R script or creating a new one, it will display in the source window (see Chapter 4.2). Some refer to this window as the 'script editor'. An R Script starts as an empty file. Good coding etiquette (see Chapter 5.5 demands that we use the first line to indicate what this file does by using a comment `#`. Here is an example for our 'TidyTuesday' R Project.

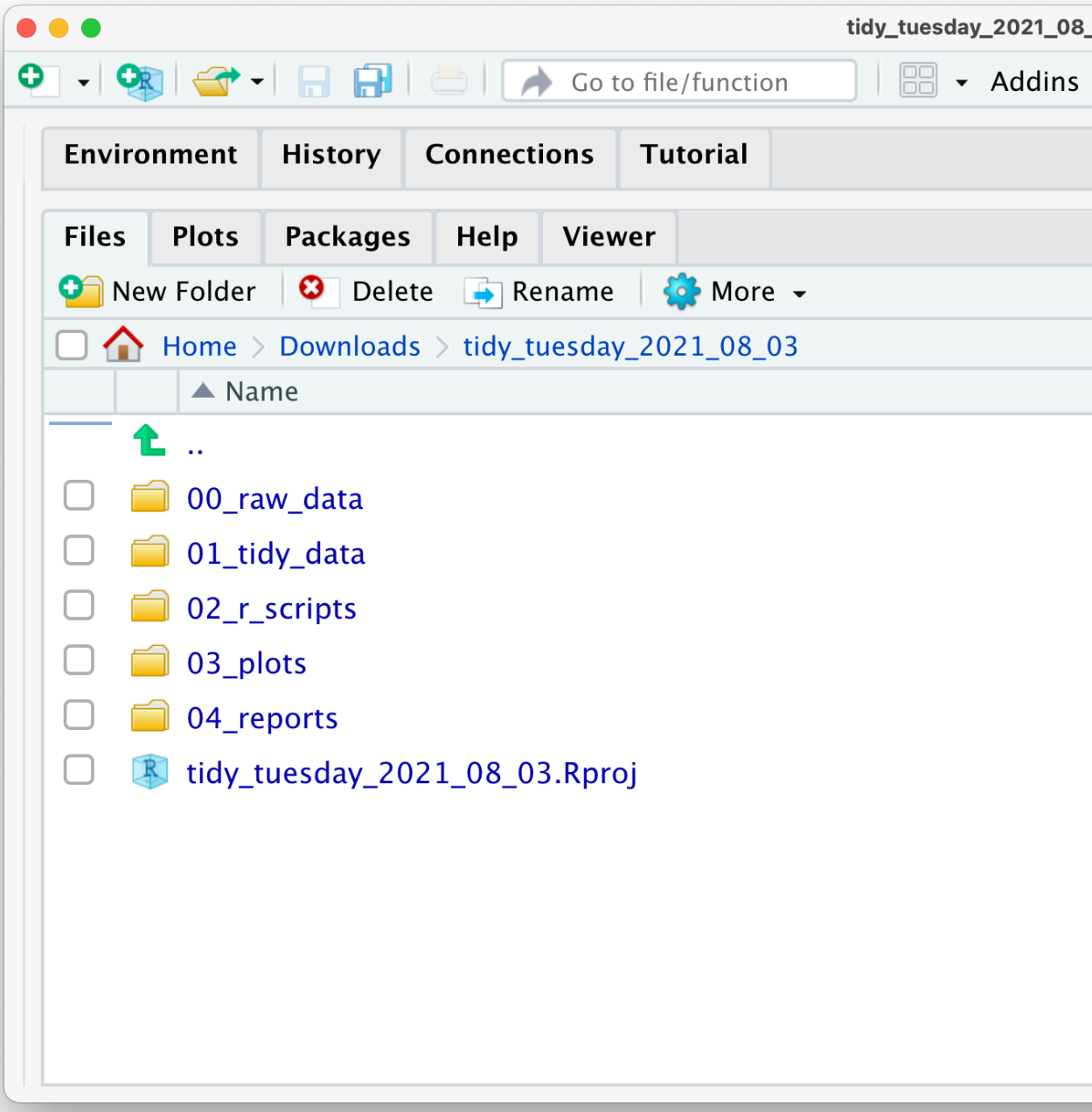
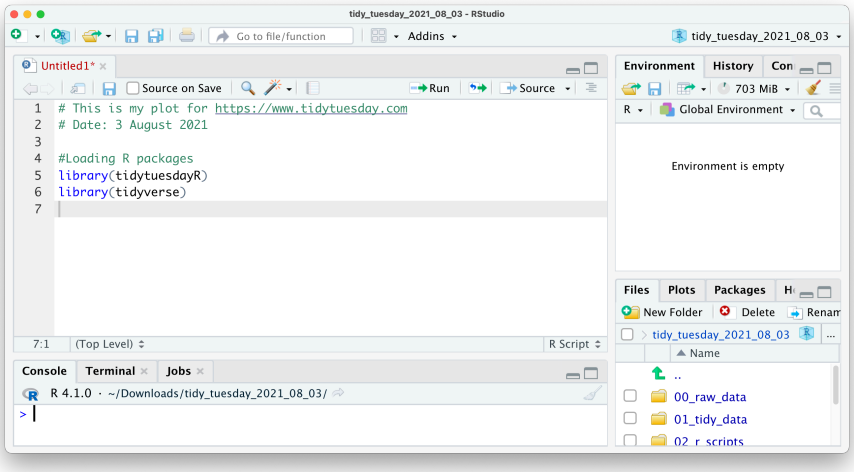
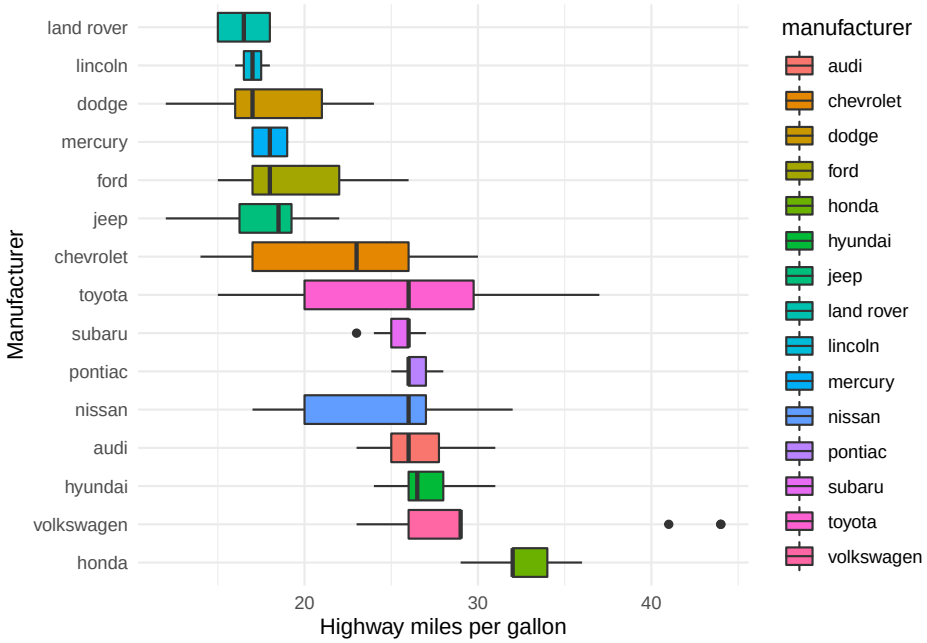


Figure 6.1: An example of a scalable folder structure for your project

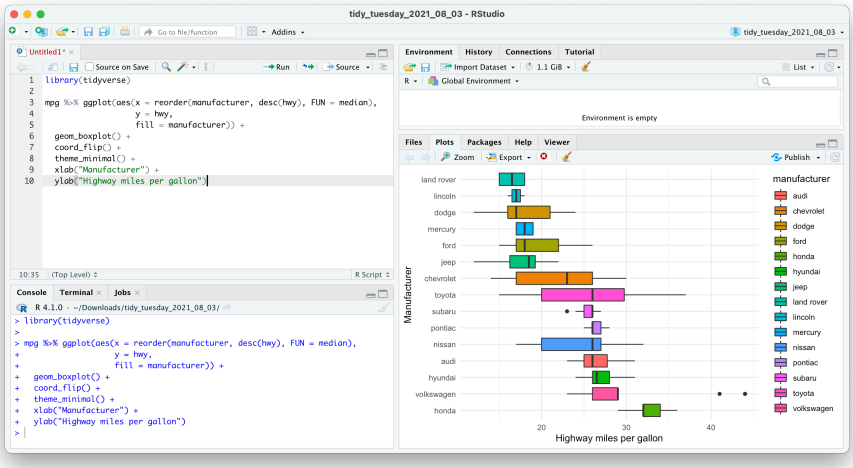


All examples in this book can easily be copied and pasted into your own R Script. However, for some code you will have to install the R package `r4np` (see above). Let's try it with the following code. The plot this code creates reveals which car manufacturer produces the most efficient cars.

```
library(tidyverse)

mpg %>% ggplot(aes(x = reorder(manufacturer, desc(hwy), FUN = median),
                   y = hwy,
                   fill = manufacturer)) +
  geom_boxplot() +
  coord_flip() +
  theme_minimal() +
  xlab("Manufacturer") +
  ylab("Highway miles per gallon")
```



You are probably wondering where your plot has gone. Copying the code will not automatically run it in your R Script. However, this is necessary to create the plot. If you tried pressing `Return` , you would only add a new line. Instead, you need to select the code you want to run and press `Ctrl+Return` (PC) or `Cmd+Return` (Mac). You can also use the `Run` command at the top of your source window, but it is much more efficient to press the keyboard shortcut. Besides, you will remember this shortcut quickly, because we need to use it very frequently. If all worked out, you should see the following:

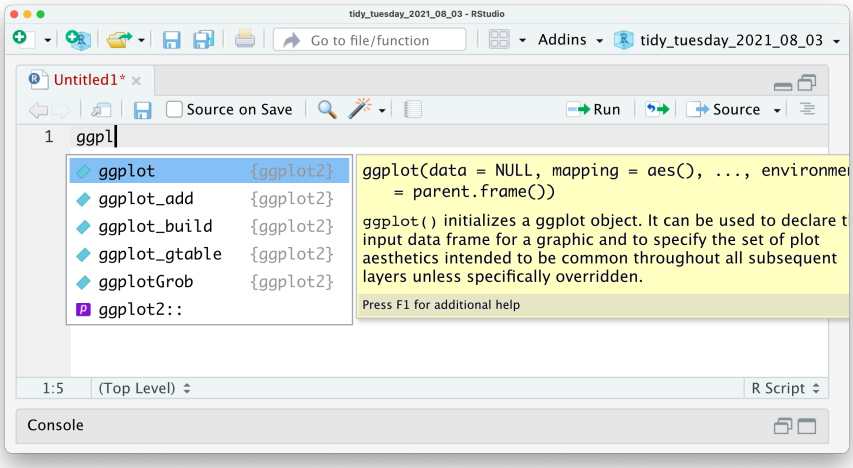


As you can see, cars from Honda appear to drive furthest with the same amount of fuel (a gallon) compared to other vehicles. Thus, if you are looking for a very economical car, you now know where to find them.

The R Script editor has some conveniences for writing your code that are worth pointing out. You probably noticed that some of the code we have pasted is blue, and some other code is in green. These colours help to make your code more readable because they carry a specific meaning. In the default settings, green stands for any values in `""`, which usually stands for `characters`. This is also called 'syntax highlighting'.

Moreover, code in R Scripts will be automatically indented to facilitate reading. If for whatever reason, the indentation does not happen, or you accidentally undo it, you can reindent a line with `Ctrl+I` (PC) or `Cmd+I` (Mac).

Lastly, the console and the R Script editor both feature code completion. This means that when you start typing a the name of function, R will provide suggestions. These are extremely helpful and make programming a lot faster. Once you found the function you were looking for, you press `Return` to insert it. Here is an example of what happens when you have the package `tidyverse` loaded and type `ggpl`. Only functions that are loaded via packages or any object in your environment pane benefit from code completion.



Not only does RStudio show you all the available options, but it also tells you which package this function is from. In this case, all listed functions are from the `ggplot2` package. Furthermore, when you select one of the options but have not pressed `Return` yet, you also get to see a yellow box, which provides you with a quick reference of all the arguments that this function accepts. So you do not have to memorise all the functions and their arguments.

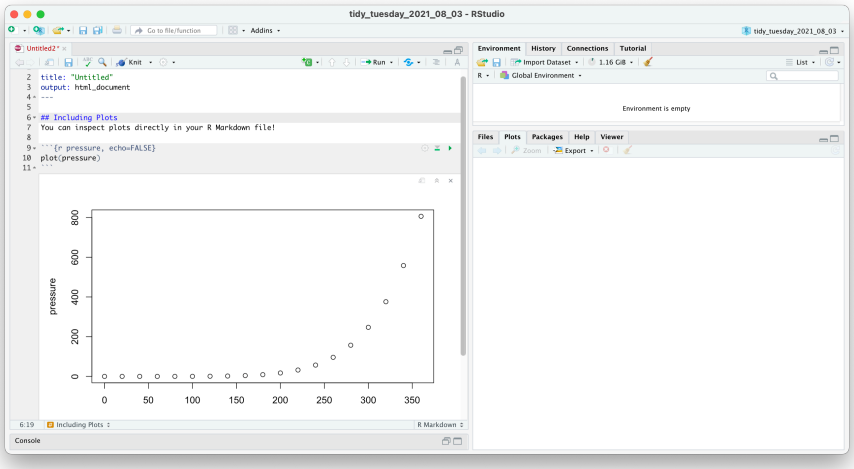
6.4 Using R Markdown

There is too much to say about R Markdown, which is why I only will highlight that they exist and point out the one feature that might convince you to choose these formats over plain R Scripts: They look like a Word document (almost).

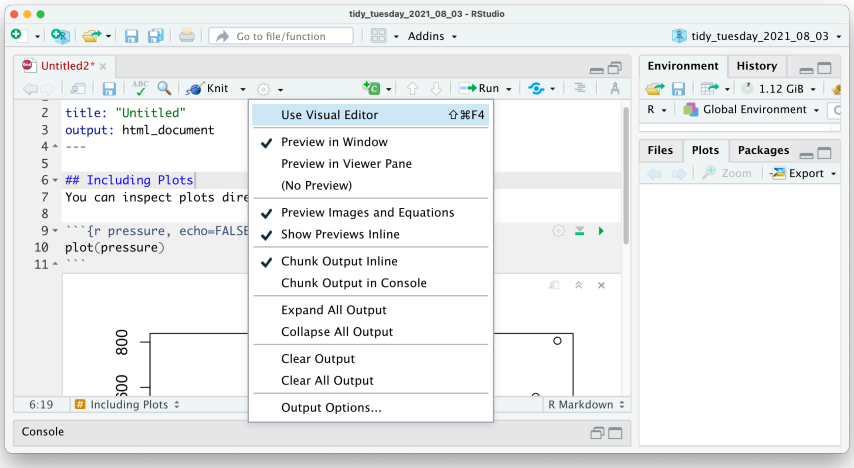
As the name indicates, R Markdown files are a combination of R Scripts and Markdown. Markdown is a way of writing and formatting text documents without needing software like MS Word. Instead, you write everything in plain text. Such plain text can be converted into many different document types such as HTML websites, PDF or Word documents. If you would like to see how it works, I recommend looking at the R Markdown Cheatsheet.

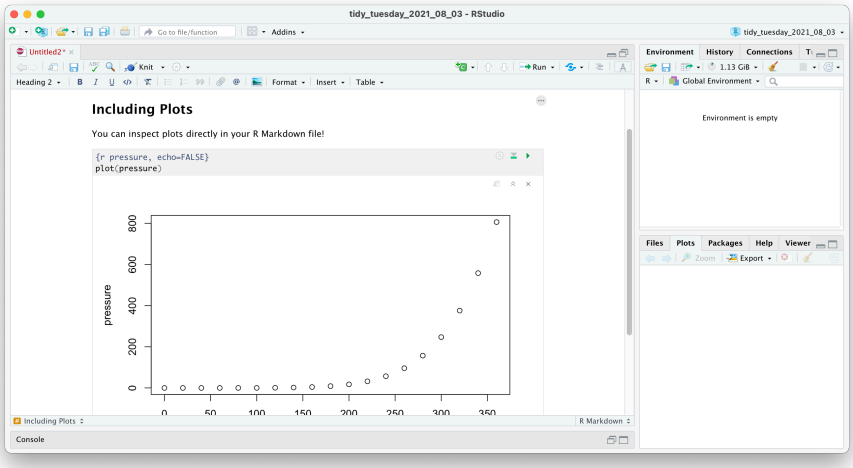
An R Markdown file works oppositely to an R Script. By default, an R Script considers everything as code and only through commenting `#` we can include text to describe what the code does. This is what you have seen in all the coding examples so far. On the other hand, an R Markdown file considers everything as text, and we have to specify what is code. We can do so by inserting 'code chunks'. Therefore, there is less of a need to use comments `#` in R Markdown files because you can write about it. Another convenience of R

Markdown files is that results from your analysis are immediately shown underneath the code chunk.



If you switch your view to the `Visual Editor`, it almost looks like you are writing a report in MS Word.





So, when should you use an R Script, and when should you use R Markdown. The rule-of-thumb is that if you intend to write a report, thesis or another form of publication, it might be better to work in an R Markdown file. If this does not apply, you might want to write an R Script. As mentioned above, R Markdown files emphasise text, while R Scripts primarily focus on code. In my projects, I often have a mixture of both. I use R Scripts to carry out data wrangling and my primary analysis and then use R Markdown files to present the findings, e.g. creating plots, tables, etc. By the way, this book is written in R Markdown using the `bookdown` package.

No matter your choice, it will neither benefit nor disadvantage you in your R journey or when working through this book. The choice is all yours. You likely will come to appreciate both formats for what they offer.

Chapter 7

Data Wrangling

You collected your data over months (and sometimes years) and all you really want to know is whether your data makes sense and reveals something nobody would have ever expected. However, before we can truly go ahead with our analysis, it is essential to understand whether our data is 'tidy'. Very often, data we receive is everything else but clean and we need to not only check whether our data is fit for analysis, but also ensure it is in a format that is easy to handle and work with. For small datasets, this is usually a brief exercise. However, I found myself cleaning data for a month, because the dataset was spread out into multiple spreadsheets (no pun intended) with different numbers of columns and odd column names. Thus, data cleaning or data wrangling is an essential first step in any data analysis. It is a step that cannot be skipped and has to be performed on every new dataset.

Luckily, R provides many useful functions to make our lives easier. If you are like me and used to do this in Excel, you will be in for a treat. It is a lot simpler using R to do achieve a clean dataset.

Here is an overview of the different steps we usually work through before we can start with our main analysis. This is list is certainly not exhaustive:

- Data import
- Checking data types
- Recoding and arranging factors, i.e. categorical data.
- Running missing data diagnostics
- and other things

7.1 Import your data

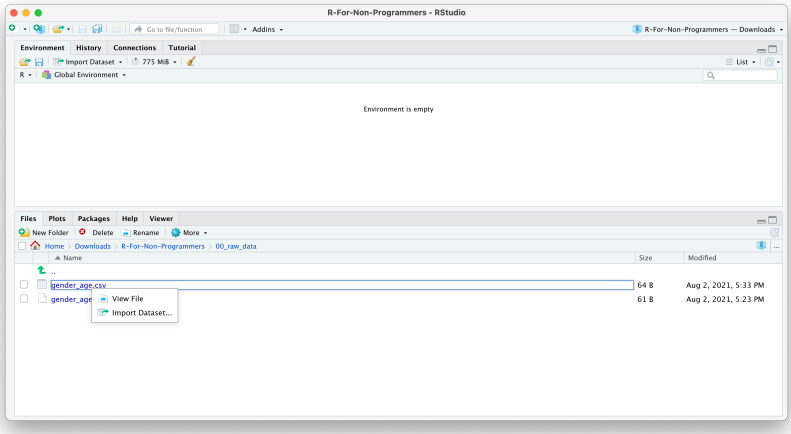
The `r4np` package hosts a number of different datasets to work with, but at some point you might want to apply your R knowledge to your own data. Therefore, an important first step is to import your data into RStudio. There are three different methods all of which are very handy:

1. Click on your data file in the Files pane and choose 'Import Dataset'.
2. Use the `Import Dataset` button in the Environment pane.
3. Import your data calling one of the `readr` functions in the console or RScript

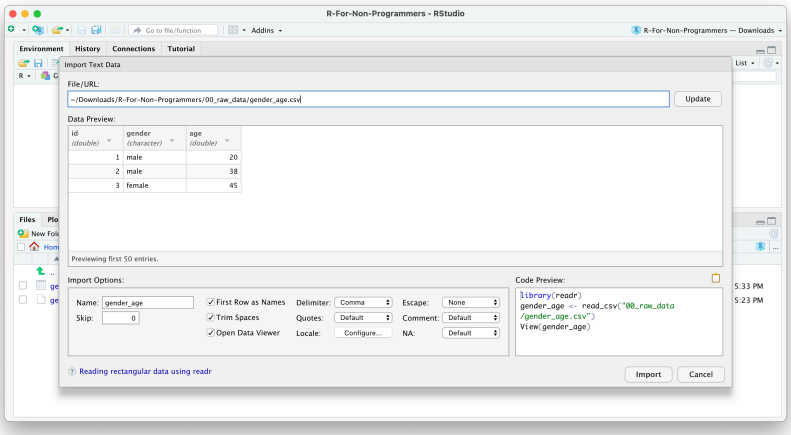
7.1.1 Import data from the Files pane

This approach is by far the easiest. Let's assume you have a dataset called `gender_age.csv` in your `00_raw_data` folder. If you wish to import it, you can do the following:

1. Click on the name of the file
2. Select `Import Dataset`.

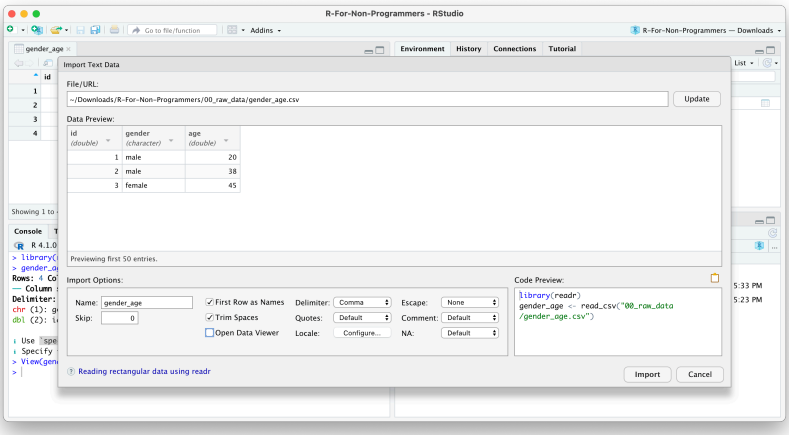


3. A new window will open and you can choose different options. You also see a little preview of how the data looks like. This is great if you are not sure whether you did it correctly.



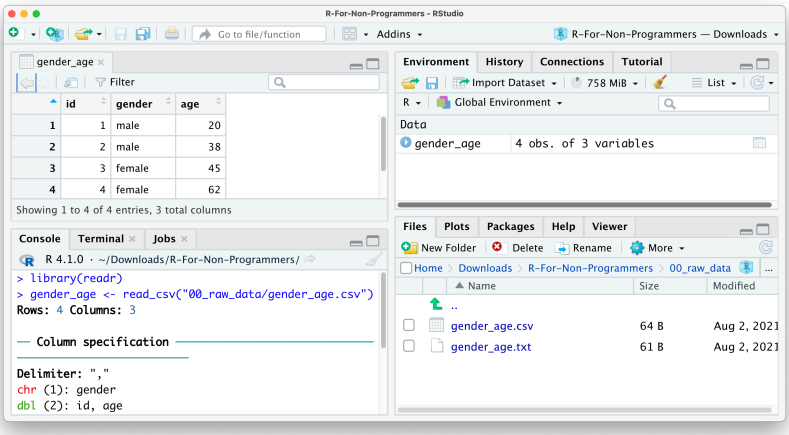
4. You can make changes to how the data should be imported, but in most cases the default should be fine. Here is quick breakdown of the most important options:

   - `Name` allows you to change the object name, i.e. the name of the object this data will be assigned to. I tend to use `df_raw` (`df` stand for `dataframe`, which is how R calls such rectangular datasets).
   - `Skip` is useful if your data file starts with a number of empty rows at the top. You can remove them here.
   - `First Row as Names` is ticked by default. In most Social Science projects we tend to have the name of the variables as the first row in your dataset.
   - `Trim Spaces` removes any unnecassry white-space in your dataset. Leave it ticked.
   - `Open Data Viewer` allows you to look at your imported dataset. I use it rarely, but it can be helpful at times.
   - `Delimiter` defines how your columns are separate from each other in your file. If it is a `.csv` it would imply it is a 'comma-separated value', i.e. `,`. This can be changed for different files, depending on how your data is delimited. You can even use the option `Other…` to specify a custom separation option.
   - `NA` specifies how missing values in your data are acknowledged. By default, empty cells in your data will be recognised as missing data.



5. Once you are happy with your choices, you can click on `Import`.

6. You will find your dataset in the Environment pane.



In the console you can see that R also provides the `Column specification`, which we need later when inspecting 'data types'. `readr` automatically imports all text-based columns as `chr`, i.e. `character` values. However, this might not be always true. More on this aspect of data wrangling in Chapter 7.4.

CONTINUE FROM HERE

To achieve this we rely on `readr` from the `tidyverse` package. It can install a range of different datatypes, including `.csv`, `.tsv`, `.txt`. If you want to import data from an `.xlsx` file you have to use another package called `readxl`.

Here are some examples of how you can use `readr` to import your data. The first part is only there to create the two

```
# Load 'readr' via 'tidyverse' or just 'readr
library(tidyverse)

# Import data from '.csv'
read_csv("00_raw_data/gender_age.csv")

# Import data from any file text file by defining the separator yourself
read_delim("00_raw_data/gender_age.txt", delim = "|")
```

7.2 Inspecting your raw data

For the rest of this chapter, we will use the `wvs` dataset from the `r4np` package. However, we do not know much about this dataset. The first way of inspecting the data is to simply type the name of the object, i.e. `wvs`.

```
# Ensure you loaded the 'r4np' package first
library(r4np)

# Show the data in the console
wvs
## # A tibble: 69,578 x 9
##    `Participant ID` `Country Code` `Country name` Gender YearOfBirth   Age
##               <dbl> <chr>          <chr>            <dbl>       <dbl> <dbl>
##  1         20070001 AND            Andorra              1        1958    60
##  2         20070002 AND            Andorra              0        1971    47
##  3         20070003 AND            Andorra              0        1969    48
##  4         20070004 AND            Andorra              1        1956    62
##  5         20070005 AND            Andorra              0        1969    49
##  6         20070006 AND            Andorra              1        1967    51
##  7         20070007 AND            Andorra              1        1985    33
##  8         20070008 AND            Andorra              0        1963    55
##  9         20070009 AND            Andorra              1        1978    40
## 10         20070010 AND            Andorra              1        1979    38
## # ... with 69,568 more rows, and 3 more variables: relationship_status <chr>,
## #   Freedom.of.Choice <dbl>, Satisfaction-with-life <dbl>
```

The result is a series of rows and columns. The first information we receive is: `A tibble: 69,578 x 9`. This indicates that our dataset has 68,578 observations (i.e. rows) and 9 columns (i.e. variables). This rectangular format is the one we encounter most frequently in Social Sciences (and probably beyond). If you ever worked in Excel this format might be quite familiar.

Even though it might be nice to look at this dataset in this way, it is not particularly useful. Depending on your monitor size you might only see a small number of columns. All in all, we hardly ever will find much use in inspecting data this way. Luckily there are other functions that can help us.

If you want to see each variable covered in the dataset and their data types, you can use the function `glimpse()`.

```
glimpse(wvs)
## Rows: 69,578
## Columns: 9
## $ `Participant ID`         <dbl> 20070001, 20070002, 20070003, 20070004, 20070~
## $ `Country Code`           <chr> "AND", "AND", "AND", "AND", "AND", "AND", "AN~
## $ `Country name`           <chr> "Andorra", "Andorra", "Andorra", "Andorra", "~
## $ Gender                   <dbl> 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, ~
## $ YearOfBirth              <dbl> 1958, 1971, 1969, 1956, 1969, 1967, 1985, 196~
## $ Age                      <dbl> 60, 47, 48, 62, 49, 51, 33, 55, 40, 38, 54, 3~
## $ relationship_status      <chr> "married", "living together as married", "sep~
## $ Freedom.of.Choice        <dbl> 10, 9, 9, 9, 8, 10, 10, 8, 8, 10, 9, 8, 10, 7~
## $ `Satisfaction-with-life` <dbl> 10, 9, 9, 8, 7, 10, 5, 8, 8, 10, 8, 8, 10, 7,~
```

The output of glimpse shows us the name of each column/variable after the `$`, for example `` `Participant ID` ``. The `$` is used to look up certain variables in our dataset. If we want to inspect only the column 'Gender' we could write the following:

```
wvs$Gender
##     [1]  1  0  0  1  0  1  1  0  1  1  0  0  1  1  1  0  0  1  1  1  1  1  1  0
##    [25]  1  0  1  1  1  0  1  0  0  1  0  0  0  1  0  1  1  1  0  0  0  1  1  1
##    [49]  1  1  1  1  1  0  0  1  1  1  1  0  1  0  0  0  0  1  1  1  0  1  0  1
##    [73]  1  1  1  1  1  0  1  1  1  1  0  1  1  1  1  1  0  1  1  1  1  1  1  1
....
```

I use `glimpse()` a lot for different purposes, for example:

- to check correctness of data types
- to inspect column names for errors
- to look up column names

`skim()`

7.3 Data cleaning

Create consistency in your column naming convention

`janitor`

7.4 Change data types and explain what they are

7.5 Recoding and rearranging factors (e.g. gender coded as 0 and 1)

7.6 Dealing with missing data

7.7 Latent constructs and their reliability

7.7.1 Theory

7.7.2 Visualisation

7.7.3 Computation

Chapter 8

Descriptive Statistics

Chapter 9

Correlations

Chapter 10

Comparing groups

Chapter 11

Regression: Creating models to predict future observations

Chapter 12

Mixed-methods research: Analysing qualitative in R

Chapter 13

Where to go from here: The next steps in your R journey

13.1 GitHub: A Gateway to even more ingenious R packages

13.2 Books to read and expand your knowledge

13.3 Engage in regular online readings about R

- Tidyverse Blog
- R-blogger

13.4 Join the Twitter community and hone your skills

- #RStats
- TidyTuesday

Chapter 14

Exercises: Solutions

14.1 Solutions for 5.6

```
#1 ---------------
sqrt(25-16)+2*8-6
## [1] 13

#2 ---------------
"Five" == 5
## [1] FALSE

#3 ---------------
books <- list("Harry Potter and the Deathly Hallows",
              "The Alchemist",
              "The Davinci Code",
              "R For Dummies")

books
## [[1]]
## [1] "Harry Potter and the Deathly Hallows"
##
## [[2]]
## [1] "The Alchemist"
##
## [[3]]
## [1] "The Davinci Code"
##
## [[4]]
## [1] "R For Dummies"
```

```
# 4 ---------------
x_x <- function(number1, number2){       # Creates a function that takes 2 arguments
  result1 <- number1*number2             # Result1 multiplies the two arguments
  result2 <- sqrt(number1)               # Result2 computes the squareroot of the 1st
  result3 <- number1-number2             # Result3 subtracts the 2nd argument from the
  return(c(result1, result2, result3))   # Prints all three results into the console
}

x_x(2,3)
## [1]  6.000000  1.414214 -1.000000

# 5 ---------------

# First, install the package.

# Second, load the package with 'library()'
#         or use '::' to load only a particular function once.

# Third, call the function you are interested in.
```

Bibliography

Cambridge Dictionary (2021). Concatenate. Accessed: 2021-07-28.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data.* " O'Reilly Media, Inc.".

Wickham, Hadley (2021). The tidyverse style guide. Accessed: 2021-08-05.